IL LINGUAGGIO MACCHINA DELLO SPECTRUM

VOL.1

EDIZIONE ITALIANA

James Walsh

GRUPPO EDITORIALE JACKSON

IL LINGUAGGIO MACCHINA DELLO SPECTRUM VOL.1

James Walsh

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

SOMMARIO

INTRODUZIONE

Lo scopo di questo libro è quello di fornire un'introduzione coerente ed elementare alla programmazione in linguaggio macchina. Tale introduzione fu ritenuta necessaria, ovvero l'autore desiderava che ne esistesse una, quando egli cominciò a cimentarsi con questo argomento alquanto misterioso. Soltanto più tardi si rese conto che in realtà non c'è alcun mistero.
In questo libro ci si è sforzati di evitare di cadere nella trappola di assumere come note certe conoscenze di base da parte del lettore, ovvero di presumere che egli non conosca per niente la materia. È stato quindi necessario affrontare l'argomento dal punto di vista dello studente, cercando sempre di anticipare i problemi che potrà incontrare. Se troverete il risultato troppo elementare, questo libro probabilmente non fa per voi; se invece vi sembrerà troppo difficile, vi faccio le mie scuse ma devo pensare che per voi non ci siano più speranze (cercate invece di essere eletti al parlamento).

CAPITOLO 1

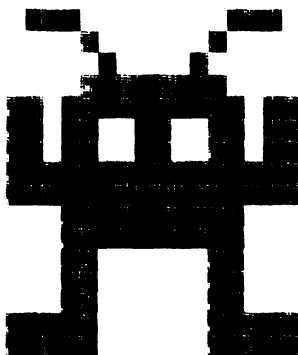
PERCHÈ IL LINGUAGGIO MACCHINA?

La risposta a questa domanda è, molto semplicemente, perché il linguaggio macchina è la lingua propria del computer; di conseguenza "essa è più comprensibile per il computer". Vi potrei offrire una descrizione più complessa e tecnicamente corretta del linguaggio macchina, ma senza dubbio vi lascerei più confusi di prima. Comunque, questo libro è un tentativo di rimuovere l'aura di mistero e la confusione che circondano il problema del cominciare con i computer, ed in particolare con il vostro Spectrum. Si è quindi cercato di fornire tutte le spiegazioni nel modo più semplice possibile. Se, quando finirete il libro avrete capito le nozioni essenziali del linguaggio macchina e sarete sufficientemente esperti da poterlo usare, allora questo lavoro non sarà stato vano.
Il massimo che i computer riescono a fare è calcolare "uno più uno" e giungere con precisione alla risposta esatta: "due". In questo non c'è niente di particolarmente intelligente, né di grandioso.
Tuttavia essi sembrano avere un'infinita capacità di intimidire gli uomini, cosicché è possibile convincersi facilmente che i computer siano troppo complicati, troppo veloci, troppo intelligenti o semplicemente spaventosi per avvicinarli. Sebbene possa apparire il contrario, il vostro Spectrum non è altro che un insieme di componenti elettronici che nel loro complesso sono capaci di elaborare, secondo le istruzioni precedentemente fornite, le informazioni, anche queste fornite dall'esterno. Il risultato finale è una macchina sofisticata, ma essa rimane sempre uno strumento da usare, senza intelligenza o volontà proprie. Esso può lavorare solo con ciò che gli viene fornito, e può fare quello che fa solo perché voi gli dite di farlo. È vero che esegue il compito in modo molto efficiente ed attendibili, e può maneggiare calcoli lunghi e complessi in una frazione piccolissima del tempo che occorrerebbe ad un cervello umano; tuttavia, come già si è detto, la sua unica vera dote ed abilità è quella di ottenere sempre e con precisione la risposta esatta alla richiesta di sommare "uno più uno".
Ma perché preoccuparsi di comprendere la natura del linguaggio macchina e poi dedicare del tempo a cercare di capire come lo si usa? Non abbiamo già una "lingua" del tutto adeguata con cui comunicare con il nostro Spectrum: il Basic? Certo, il Basic è una lingua che vi sarà ormai familiare e almeno sembra abbastanza comprensibile.

Infatti essa è proprio questo: una lingua per computer studiata per gli uomini. Il suo scopo è di permetterci di parlare con il no stro ZX80/Zx 81/Spectrum in modo conciso ed ordinato, usando una forma che è comprensibile per noi e che può essere "interpretata" dal computer. Proprio così, "interpretata". Nella memoria del computer c'è un vocabolario completo, dal Basic al linguaggio macchina. Il computer si basa esclusivamente su di esso per tradurre e comprendere tutto ciò che gli viene detto in Basic, prima di poter eseguire le istruzioni fornite. Se quindi lo scopo è quello di ottenere una migliore comprensione da parte dello Spectrum ed una sua più veloce risposta, allora diventa importante essere capaci di "parlare la sua lingua". È semplice capire quanto sarebbe difficile comunicare con qualcuno, incontrato in un paese straniero, il quale sappia parlare soltanto la sua lingua e di cui voi non conoscete nemmeno una parola. Se non si facesse uso di un qualche linguaggio a gesti abbastanza intelligente, sarebbe virtualmente impossibile fare alcun progresso. In effetti il ricorrere ai segni sarebbe soltanto uno sforzo comune per trovare un semplice linguagio che entrambi possano capire. Se invece si potesse far uso di un interprete, allora il processo comincerebbe ad essere più semplice. Comunque, resta il fatto che tutto ciò che vi dite dovrebbe essere tradotto e ritradotto, e così la conversazione risulterebbe limitata e molto lenta.
Questa è dunque la situazione dello Spectrum: esso ha un interpete incorporato. Tutto ciò che viene immesso in Basic viene interpretato nella sua lingua e viceversa, e percié c'è un limite naturale all'entità della conversazione e alla velocità con cui si può comunicare. Quando i programmi vengono scritti in Basic, lo Spectrum deve, prima di eseguirli, memorizzare e poi interpretare tutti i comandi e i dati. L'interpretazione in linguaggio macchina è necessaria per consentire al microprocessore Z80 di funzionare, dato che questo è l'unico linguaggi che esso è in grado di capire.
Inevitabilmente, questo processo di ricerca nella memoria del computer, di reperimento delle istruzioni, dei dati e di esecuzione richiede tempo. Se poi tutto ciò deve essere anche tradotto dal Basic al linguaggio macchina prima di ogni altra cosa, il tempo necessario sarà molto superiore. Ci sono casi in cui questa può essere una considerazione importante, per esempio quando si richiede controllo della grafica o rapidità di risposta del programma.
Come quando si viaggia all'estero, se alla fine si vuole essere in grado di comunicare speditamente, con facilità ed accuratezza, non c'è altro da fare che imparare la lingua del paese in cui si viaggia. Avete scelto di viaggiare nel territorio proprio al computer e dovete quindi imparare la sua lingua.
Può sembrare poco ragionevole, ma state certi che non sarà lo Spectrum ad imparare la vostra, non ancora almeno! Chiaramente imparare una nuova lingua straniera non è del tutto facile. Per fortuna il linguaggio del computer è estremamente semplice: il modo in cui poi il computer lo usa è una cosa molto diversa, ma cercheremo di capirla insieme. Il computer è bravo a sommare "uno più uno", e poiché ciò è l'unica cosa che sa fare, passa tutta la sua esistenza ad occuparsi di "zeri" e di "uni". Forse è utile menzionare, a questo punto, che in tutto il libro

si userà la convenzione di scrivere zero con "0" per evitare confusione con la lettera "o". Per capire come mai il computer si occupi solo di "zeri" e di "uni", e come ciò possa permettergli di eseguire compiti anche molto complessi è necessario avere una certa conoscenza di come sia costituito il computer e di come riesca a funzionare. Prima di fare questo vediamo brevemente qual'è la velocità con cui esso risponde alle istruzioni in linguaggio macchina rispetto a quelle in Basic.
Inserite questo programma:

```
   5 BRIGHT 1: PAPER 4: BORDER 6
: INK 1: CLS
  10 PRINT AT 5,8;"
  15 PRINT AT 10,0;"
  20 PRINT AT 13,0;"
  25 PRINT AT 0,0;"MEMORIZZAZION
E IMMAGINE"
  30 FOR x=0 TO 7167
  40 POKE (25600+x),PEEK (16384+
x)
  50 NEXT x
  60 PRINT AT 0,0;"MEMORIZZAZION
E COMPLETA"
  70 PAUSE 30
  80 CLS
  90 PRINT "RECUPERO IMMAGINE
"
 100 FOR x=0 TO 7167
 110 POKE (16384+x),PEEK (25600+
x)
 120 NEXT x
 130 PRINT AT 0,0;"FINE PROGRAMM
A"
```

MEMORIZZAZIONE IMMAGINE

È lento, non vi pare? Adesso inserite l'equivalente versione del programma scritta in linguaggio macchina, come descritta sotto, e vedete la differenza. Cancellate le righe 30, 40, 50, 70, 100, 110, 120 e aggiungete le seguenti:

```
  30 RANDOMIZE USR 25501
  80 PAUSE 100
 100 RANDOMIZE USR 25513
 150 STOP
 200 FOR a=25501 TO 25524: READ
a$
 210 LET ans=(CODE a$(1)-48)*16
 220 IF ans>9*16 THEN LET ans=an
s-7*16
 230 LET l=CODE a$(2)-48: IF l>9
 THEN LET l=l-7
 240 LET ans=ans+l: POKE a,ans
 250 NEXT a
 260 RETURN
 300 DATA "21","00","40","11","0
0","64","01","00","1C","ED","B0"
,"C9"
 310 DATA "21","00","64","11","0
0","40","01","00","1C","ED","B0"
,"C9"
```

Adesso aggiungete questa riga:

```
1000 CLS : PAUSE 30: RANDOMIZE U
SR 25513: PAUSE 30: GO TO 1000
```

ed eseguite.

```
RUN 1000
```

Vi siete convinti adesso?

Torniamo ora al modo in cui funziona il computer. Essenzialmente esso compie i calcoli in modo aritmetico, usando il sistema binario ed una memoria in cui conserva le informazioni. Il vostro Spectrum come computer digitale riconosce solo DUE stati: esso è capace soltanto di distinguere la presenza e l'assenza di un impulso elettrico. Esso opera quindi in base alla sua condizione di "acceso" o "spento": da qui deriva il termine "sistema binario" e la natura "a due stati" del computer. Con questo elemento importante, passiamo a vedere come rappresentare questi due stati per mezzo di "zero" e "uno".
È una progressione logica rappresentare "assenza di segnale elettronico" con "0" e "presenza di segnale elettrico" con "1". I due stati stabili del computer possono essere così rappresentati:

1. C'è un segnale? Sì -rappresentato con "1"
2. C'è un segnale? No -rappresentato con "0".

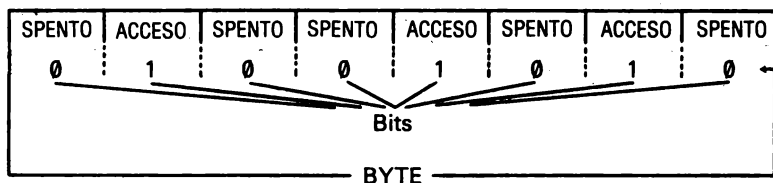
Il paragone più semplice da fare è quello con il comune interruttore, che può essere nella posizione di "acceso" o in quella di "spento". Quando l'interruttore è acceso, la corrente elettrica può scorrere; quando è spento, la corrente non passa.
Può sembrare che adesso abbiamo ridotto il computer a qualcosa che può contare almeno fino a due, a qualcosa di simile al comune interruttore della luce. Ciò può effettivamente essere vero solo fino ad un certo punto, ma chiaramente porterebbe a sottostimare il vostro povero computer. In effetti, il vostro Spectrum, o meglio il suo microprocessore Z80 ha la capacità di mettere insieme una grandissima serie di queste decisioni di "acceso/spento" e produrre così un gran numero di combinazioni diverse, ciascuna delle quali ha un suo significato particolare. Per comprendere meglio tutto ciò considerate ad esempio due di questi semplici interruttori, che possono essere in posizione di "acceso" o di "spento" e combinateli. In questo modo si possono ottenere quattro diverse combinazioni:

1. acceso e acceso ----diventa 1 1
2. acceso e spento ----diventa 1 0
3. spento e acceso ----diventa 0 1
4. spento e spento ----diventa 0 0

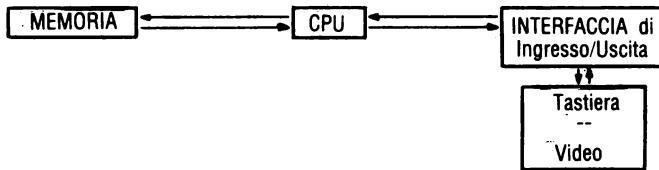
Mettendo insieme tre di questi "interruttori", ancora rappresentativi degli stati "acceso/spento" riconosciuti dal computer, si ottengono otto stati diversi:

1. spento e spento e spento --- 0 0 0
2. spento e spento e acceso --- 0 0 1

3. spento e acceso e spento --- 0 1 0
4. spento e acceso e acceso --- 0 1 1
5. acceso e spento e spento --- 1 0 0
6. acceso e spento e acceso --- 1 0 1
7. acceso e acceso e spento --- 1 1 0
8. acceso e acceso e acceso --- 1 1 1

Ciò che adesso dovrebbe cominciare ad essere evidente è la semplice regola dele "combinazioni". Se noi prendiamo due stati specifici, essi possono essere combinati in un massimo di 2×2 (cioè quattro) modi diversi. Prendendo tre come punto di partenza, un massimo di 2×2×2 (cioè otto) combinazioni diverse divengono possibili. Ora, nello stesso modo in cui possiamo mettere insieme le lettere dell'alfabeto e formare così combinazioni specifiche che noi riconosciamo come parole, ciascuna con un suo significato particolare, così il computer può considerare le combinazioni delle due cifre "0" e "1" e produrre delle "parole" singolarmente identificabili. Il vostro Spectrum può maneggiare "parole" di otto cifre, e usando la regola della combinazione si vede che ciò consente di avere un "vocabolario" di 256 parole diverse (2×2×2×2×2×2×2×2, ovvero "due all'ottava"). Combinando le otto cifre in questo modo, si creano delle "parole" che il computer è in grado di capire; ogni "parola" viene chiamata "numero binario". Il nome che si usa per indicare questo numero binario è BYTE, mentre le cifre di cui esso è composto sono dette BIT.

In questo capitolo introduttivo si è cercato di aiutare voi lettori a cominciare a capire sia la semplicità che la sofisticazione del vostro computer. Il linguaggio macchina è la lingua che esso usa e capisce, e avete visto come possa essere ridotto a serie di "zeri" e di "uni". Il prossimo capitolo sarà centrato sul modo in cui sono organizzati i vari componenti del computer, come punto di partenza, almeno a grandi linee, per capire qualcosa del modo in cui funziona e si utilizza con efficacia l'"aritmetica binaria". Il compito successivo sarà quello di vedere come le istruzioni vengano immagazzinate prima di essere passate al computer in parole che esso può comprendere e che noi possiamo utilizzare. E sebbene sarà necessario occuparci del problema della nostra interpretazione di un numero binario, non dovrete invece rassegnarvi alla terribile prospettiva di leggere interminabili sequenze di "zeri" e di "uni" o di dover assoggettare a questa dieta il vostro computer.

CAPITOLO 2

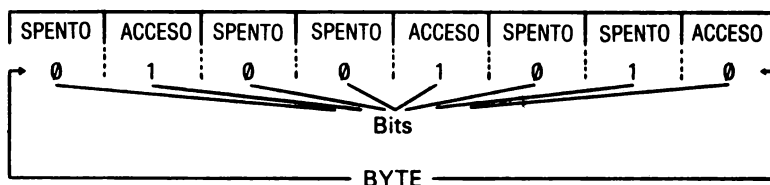
PAROLE COME NUMERI

Il disegno qui sopra è una rappresentazione schematica di come è organizzato lo Spectrum. Si è già fatto riferimento al microprocessore Z80 ed alle sue funzioni di SEARCH (ricerca), LOCATE (individuazione) ed EXECUTE (esecuzione). Questo, detto anche Unità Centrale di Elaborazione (Central Processing Unit, CPU), è la componente centrale del computer. Esso è collegato in due sensi con le altre componenti: l'interfaccia e la memoria.

L'interfaccia: consente alla CPU di far vedere i risultati delle sue fatiche passandoli in uscita ad altri dispositivi, come ad esempio il video del televisore, con cui quei risultati vengono mostrati visivamente. Alternativamente, esso consente alla CPU di conservarli in una memoria permanente su una cassetta audio di un registratore a nastro, oppure con un più raffinato sistema di conservazione, per esempio su disco. L'interfaccia è anche in grado di consentire l'immissione di informazioni (INPUT) da fuori del computer: dalla tastiera o da una memoria esterna. Ad esempio si può usare l'interfaccia per collegare e far funzionare dei circuiti contenenti sensori e permettere così al computer di eseguire svariate operazioni quali il controllo della produzione in una fabbrica o l'interruzione del vostro gioco di Space Invaders per farvi sapere che vi si è appena bruciata la cena!!! Dunque questa interfaccia è nota come il dispositivo di ingresso/uscita (INPUT/OUTPUT) del computer.

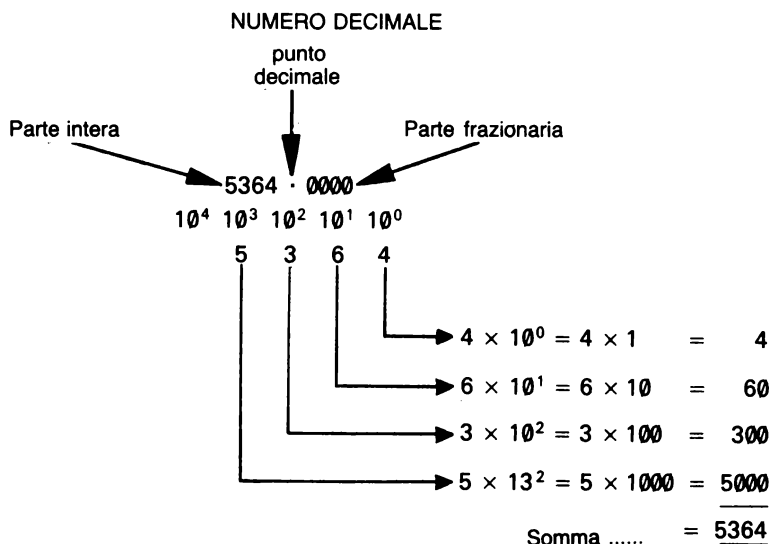
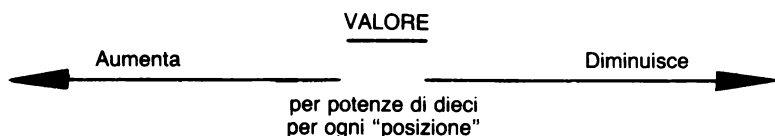
L'altra componente è la memoria.
Questa funziona proprio come ci si attenderebbe: informazioni e dati vengono immagazzinati e possono essere successivamente recuperati.
Poiché essa conserva fedelmente tutto ciò che viene immesso, è alla memoria che la CPU si rivolge quando deve trovare ciò di cui ha bisogno. I tipi di memoria sono principalmente due: "RAM" e "ROM".

"RAM" è l'abbreviazione di "Random Access Memory" (memoria ad accesso casuale). Ciò significa che la lettura o la scrittura della prima o dell'ultima parte della memoria richiedono esattamente lo stesso tempo, a differenza di quanto avviene con altri dispositivi di conservazione, come i nastri a cassetta o i dischi magnetici, nei quali per reperire l'ultima informazione è necessario prima passare attraverso tutte le precedenti informazioni. Analogamente, con un nastro audio è possibile sia scrivere, o mettere in memoria, su una RAM che leggere da essa.

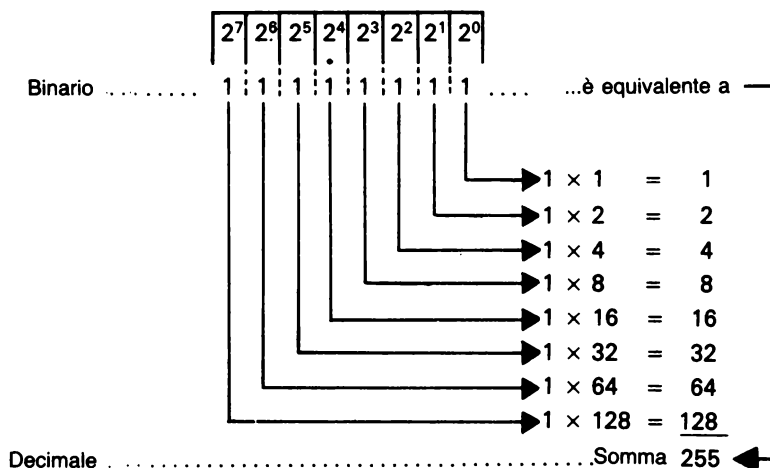
Invece ROM vuol dire "Read Only Memory" (memoria per sola lettura). Chiaramente questo indica che si può solo leggere ciò che è contenuto in quella parte della memoria, senza la possibilità di modificarla in nessun modo. Ciò è del tutto analogo al modo in cui si possono leggere le informazioni contenute in un disco musicale (farlo suonare), mentre è impossibile aggiungere nuove informazioni (tranne che con atti di vero vandalismo!!). Naturalmente potete provarci ma sarebbe proprio come se premeste il tasto di registrazione di un registratore senza inserirvi il nastro: gli ingranaggi girerebbero ma non succederebbe nulla. Per essere più precisi, è superfluo dire che le ROM sono anch'esse memorie ad accesso casuale, esattamente come le RAM: la differenza è che le ROM dovrebbero essere più correttamente chiamate "Random Access Read Only Memory" (memoria ad accesso casuale per sola lettura).

Finora ho fatto di tutto per evitare il più tradizionale "approccio numerico" ai computer e all'informatica, sperando di mantenere vivo il vostro interesse tuttavia i numeri hanno il loro posto di diritto in questi discorsi e non possono essere ignorati. Rivolgendo ora l'attenzione al modo in cui i numeri sono memorizzati ed utilizzati otterremo il punto di partenza essenziale per comprendere come usare e comunicare con lo Spectrum come un insieme integrato, non semplicemente come un assortimento di componenti separate. Ritornando all'analogia con gli interruttori del tipo acceso/spento:

Si può vedere che questo numero binario 01001010 è una specifica "parola binaria" che sta nell'interno della gamma di 256 parole del vocabolario dello Spectrum. Gli estremi di tale gamma sono rappresentati da "tutti spenti" e "tutti accesi".

Ma perché una stringa di otto zeri deve essere "zero" e una di otto uno deve essere equivalente al numero decimale 255? La risposta sta nel tipo di aritmetica usata dal computer, l'aritmetica binaria, che può forse essere meglio capita esplorando prima la struttura del sistema "decimale" con cui abbiamo più familiarità.

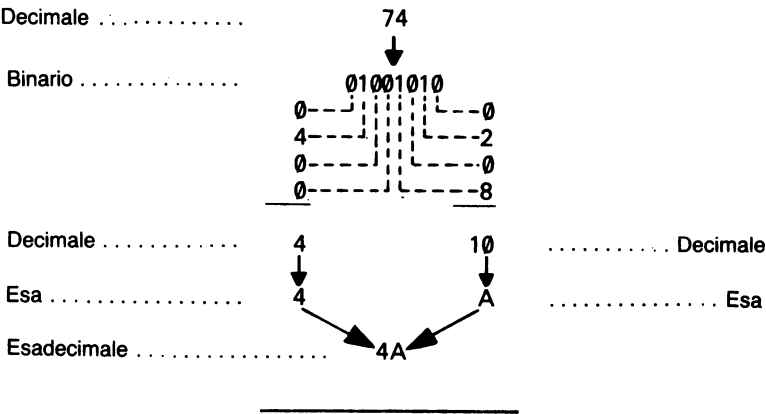
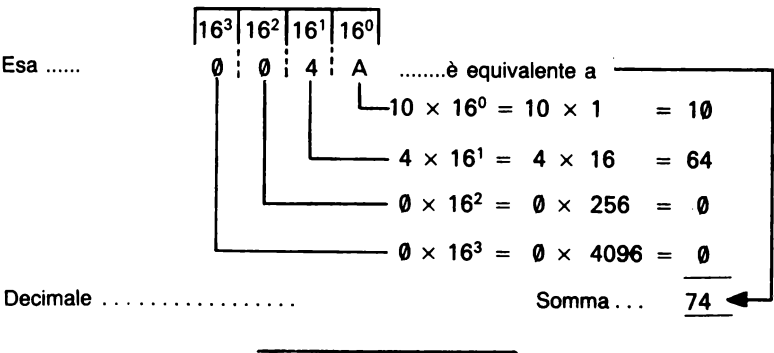
Il sistema decimale si basa su dieci cifre (da 0 a 9), e viene quindi anche chiamato sistema a "base dieci". Esso inoltre fa uso del sistema di "notazione posizionale": questo sistema fa sì che il valore di ogni cifra di un numero sia calcolato in base alla sua posizione rispetto al punto decimale. Con maggiore precisione si deve dire che la notazione posizionale si basa sulla posizione di ogni cifra rispetto al "punto di unità", non al punto decimale in quanto tale. Per esempio nell'aritmetica binaria esso diventa il "punto binario". Il "punto di unità" è quel punto che separa la parte intera da quella frazionaria di un numero. Rimanendo per il momento nell'ambito del sistema decimale, il valore di una cifra in un numero è il prodotto di sé stessa con la "potenza di 10" determinata dalla sua posizione rispetto al punto decimale. Le cifre del numero sono disposte "per colonne", e ciascuna colonna ha un valore pari a dieci volte quello della colonna immediatamente alla sua destra.

Ciascuna cifra (colonna) ha una "potenza" dieci volte maggiore di quella cifra (colonna) alla sua destra.

Il sistema binario è invece basato su due sole cifre (0 e 1), ed è quindi un sistema a "base due". Come per quello decimale anche qui si usa il sistema della notazione posizionale; qui però ogni cifra del numero binario ha un valore che ha una potenza di due volte maggiore di quella immediatamente alla sua destra.

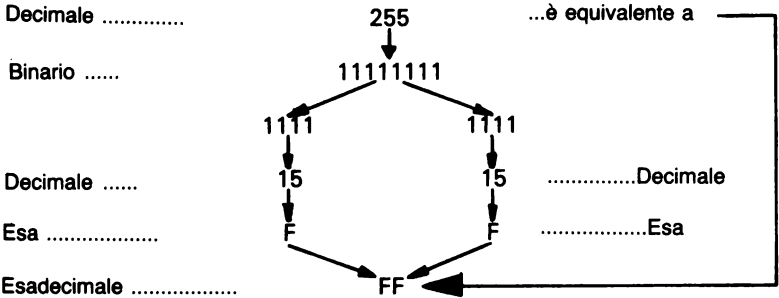
Si può vedere così che nell'esempio di prima il numero 01001010 è l'equivalente del numero decimale 74. Analogamente 00000000 = zero (decimale) e 11111111 = 255 (decimale).

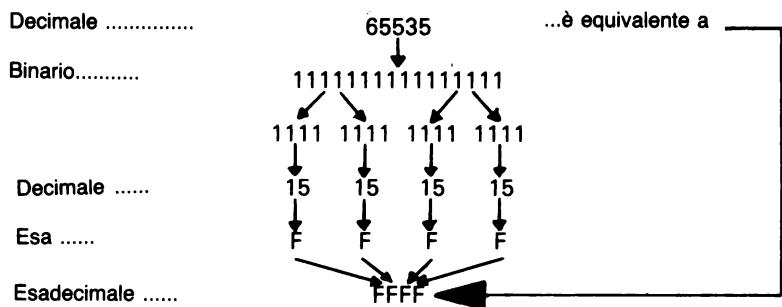
Sebbene per lavorare il computer usi l'equivalente binario dei numeri decimali, vi è stato premesso nel capitolo precedente che non sarà necessario che ricordiate o facciate uso di lunghe serie di zeri e di uno per riuscire a comunicare con il vostro Spectrum. Tuttavia non potete nemmeno far conto di poter utilizzare solamente i ben familiari numeri decimali. Comunque, mentre la "base 10" è di scarsa utilità e la "base 2" è poco pratica, la "base 16" è di eccezionale convenienza. Non disperate: è veramente più semplice di quanto sembri a prima vista. Applicando i principi delineati sopra, il sistema numerico a base 16,

detto anche "sistema esadecimale", utilizza 16 cifre. Le prime dieci della serie corrispondono a quelle del sistema decimale (da 0 a 9); le rimanenti sei sono rappresentate invece dalle prime sei lettere dell'alfabeto: A, B, C, D, E, F. La tabella qui di seguito mostra le corrispondenze fra i tre sistemi binario, ed esadecimale.

Numero binario	*Equivalente decimale*	*Equivalente esadecimale*
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Una tabella completa delle equivalenze binario-decimale-esadecimale da 0 a 255 è contenuta nell'appendice A.

Riprendiamo adesso il precedente esempio del numero 74 (decimale) ed il suo equivalente 01001010 (binario), e si noti il conveniente metodo di conversione al sistema esadecimale (spesso abbreviato a "esa" per comodità):

Numero binario	*Equivalente decimale*	*Equivalente esadecimale*
01001010	74	4A

Se operiamo allo stesso modo per il numero 255, che è il più alto considerato dalla CPU, possiamo notare la facilità con cui ogni numero binario di 8 bit possa essere rappresentato con un numero esadecimale di due cifre:

Numero binario	*Equivalente decimale*	*Equivalente esadecimale*
11111111	255	FF

Questo raggruppamento delle cifre binarie a quattro a quattro e la successiva combinazione dei loro equivalenti esadecimali è un metodo ancora più importante e molto conveniente per la rappresentazione degli "indirizzi" o "locazioni di memoria". Si è molto accentuato finora il fatto che 255 è il più alto numero che si possa utilizzare. Questa restrizione non può però essere accettata, e mostrerò tra poco il modo in cui la CPU del vostro Spectrum sia capace di utilizzare la sua abilità a maneggiare 256 parole binarie unitamente alla sua abilità a calcolare "uno più uno", ed arrivare così al numero 65536!!! Per poter trattare numeri con estensione da 0 a 65535 sono necessari numeri binari di 16 cifre, di cui il più elevato sarà 1111111111111111, ossia appunto l'equivalente di 65535 (controllate se avete dei dubbi). Tutti i numeri possibili, sia in binario che in decimale, cominciano ora a diventare piuttosto grandi e disagevoli. Il particolare valore del sistema esadecimale diventerà ora più evidente, perlomeno perché consente di riorganizzare un qualsiasi numero binario in gruppi di quattro cifre e di rappresentare ciascuno di questi gruppi con una cifra esadecimale: queste poi vengono combinate fra loro per fornire l'equivalente esadecimale a quattro cifre di un numero binario di 16 bit.

Numero binario	Equivalente decimale	Equivalente esadecimale
1111111111111111	65535	FFFF

Ogni numero nell'intervallo da 0 a 65535 può ora essere ridotto a un numero esadecimale a quattro cifre nell'intervallo da 0000 a FFFF.

Come tutti i sistemi di numerazione, i calcoli aritmetici possono essere eseguiti con l'"aritmetica binaria" e naturalmente il computer è particolarmente adatto a fare con precisione proprio questo tipo di calcoli. Tutto viene effettuato sotto forma di addizione o di sottrazione: il vostro Spectrum riesce persino a fare le sottrazioni per mezzo dell'addizione, ma di ciò parleremo meglio in seguito. Il vantaggio è naturalmente quello che in questo modo non ci dobbiamo preoccupare troppo delle moltiplicazioni e delle divisioni. In generale, le convenzioni (o regole) che si riferiscono all'aritmetica decimale valgono pure per l'aritmetica binaria. Così, il più alto numero che si può trovare in ogni colonna è uno; cioè un numero in meno del numero totale di cifre ammesse nel sistema di numerazione (2–1). Nel sistema decimale la cifra più elevata è 10–1 =9. Quando si eseguono addizioni di due numeri binari, la familiare regoletta di "riportare 1 alla colonna successiva" viene applicata quando la somma di due cifre è maggiore di 1. Ciò può essere più chiaro facendo riferimento alla seguente tabella:

Ø+Ø = Ø
Ø+1 = 1
1+Ø = 1
1+1 = 1Ø ("zero, riporto uno") ')

+	Ø	1
Ø	Ø	1
1	1	1Ø

Un esempio di addizione di due numeri binari sarà così come segue:

	01001010	(Decimale 74)
	+ 00111010	(Decimale 58)+
Riporto	0111010–	=
Somma	10000100	(Decimale 132)

Applicando gli stessi principi alla sottrazione (tranne che questa volta prendiamo a prestito due invece di riportare uno), si ha essenzialmente il rovescio della procedura di addizione.

	10000100	(Decimale 132)
	– 00111010	(Decimale 58)–
Prestito	01111010–	=
	01001010	74 decimale)

Questa è in realtà una procedura più complessa del necessario: infatti le sottrazioni vengono effettuate meglio usando un metodo noto come "il complemento a due". Questo è quanto viene effettivamente fatto dal computer. Nell'esempio di prima il risultato della sottrazione può essere ottenuto più facilmente sommando "il complemento a due" di 00111010 (il sottraendo) a 10000100 (il minuendo), per trovare il complemento a due di un qualsiasi numero binario basta cambiare tutti gli zeri in 1 e tutti gli uno in 0, e poi aggiungere 1 al risultato:

Le otto cifre meno significative sono le stesse di prima (il numero 74 decimale), mentre adesso c'è una cifra in più di "riporto", nota anche come "bit del segno". Questa cifra può essere tracurata per quanto riguarda la determinazione del

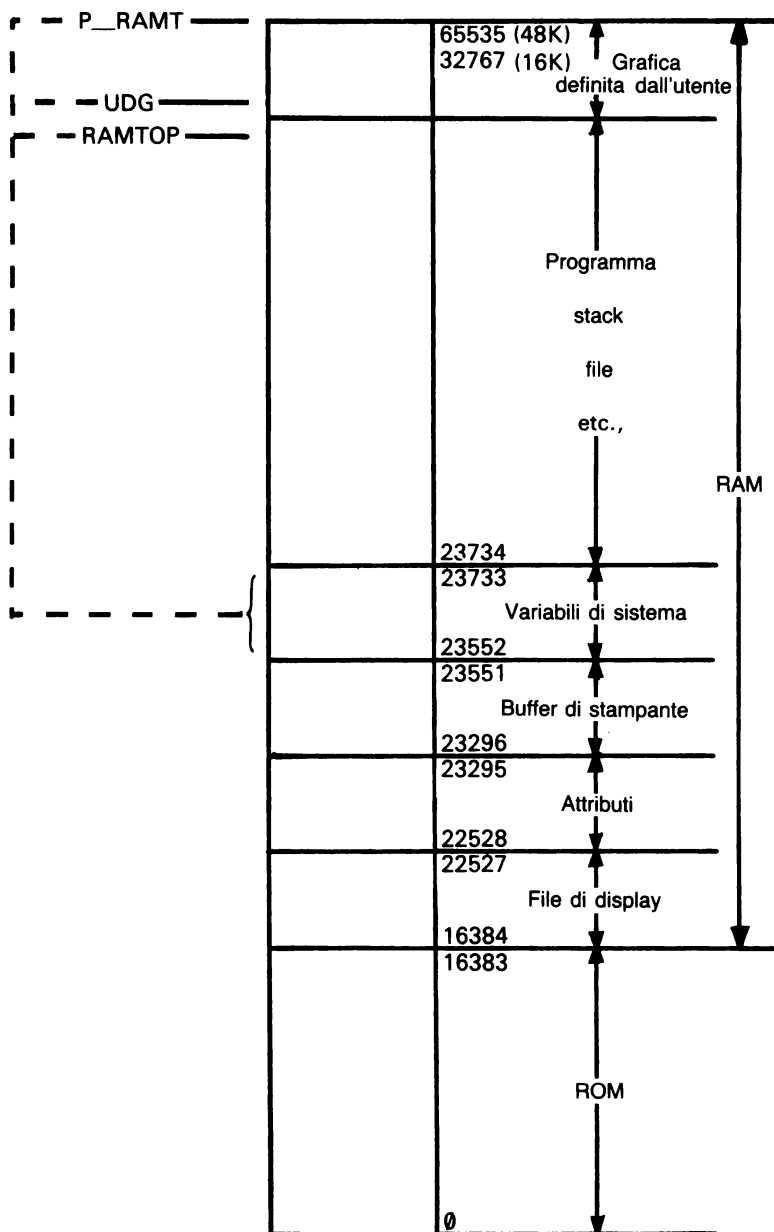
risultato numerico, ma viene preso come indicatore del fatto che il risultato del calcolo sia positivo o negativo. Quando il bit di riporto è 1 allora il numero risultante ha valore positivo; viceversa, quando il bit di riporto è 0 il risultato del calcolo è negativo.

CAPITOLO 3

GLI INDIRIZZI ED IL MODO DI RAGGIUNGERLI

Vale la pena a questo punto riassumere quanto si è detto finora.
Abbiamo cominciato con il descrivere molto generalmente il linguaggio macchina come la lingua propria del computer ed abbiamo considerato i benefici che possono derivare dall'apprendimento di tale lingua. Analogamente, partendo da una visione generale di come funziona lo Spectrum, abbiamo visto più in dettaglio come le sue parti siano collegate tra loro ed il modo in cui esse facciano uso dell'aritmetica binaria per funzionare. Ciò ha richiesto che noi affrontassimo anche un altro sistema di numerazione, il sistema esadecimale, in modo da imparare come ridurre un numero binario di otto cifre ad un numero esadecimale di due cifre, oppure un numero binario di sedici cifre ad uno esadecimale di quattro cifre. Dopo aver parlato con particolare attenzione del vocabolario dello Spectrum limitato a 256 "parole", ho fatto notare la sua abilità di conservare informazioni in oltre 65000 locazioni di memoria distinte. Questo sistema di "indirizzamento" verrà studiato molto attentamente nel corso di questo capitolo. Esamineremo inoltre nei dettagli il modo in cui la memoria dello Spectrum è strutturata ed organizzata, e cominceremo ad imparare come manipolare il contenuto di tale memoria. Il nostro primo compito è quello di esplorare la struttura e l'organizzazione della memoria dello Spectrum, facendo riferimento alla "mappa della memoria" (vedi fig. 3.1.). Ciascuna locazione di memoria è un posto in cui è possibile conservare un'informazione, ed è usualmente chiamata "indirizzo". Questo termine non è stato trovato per caso, bensì è l'uso appropriato di una parola e di un concetto già familiare a noi tutti. A ciascuna locazione viene assegnato un numero che va da 0 a 65535: tale numero diventa così l'indirizzo specifico di quella locazione. La mappa di memoria della fig. 3.1. mostra che i primi 16384 (ovvero 16K) indirizzi sono assegnati alla ROM. Questa, come già sappiamo, è la memoria per sola lettura, e consiste in pratica di un blocco di 16384 locazioni nelle quali le informazioni sono state immesse al momento della produzione del chip. Tale blocco quindi non è possibile di modifiche od alterazioni da parte dell'utente. Nella versione a 16K dello Spectrum, ci sono poi altre 16384 locazioni di memoria RAM; la versione a 48K, invece, utilizza il massimo numero di locazioni indirizzabili dalla CPU Z80; quindi, oltre ai 16K di ROM ci sono altri 48K di RAM (locazioni dalla 16384 alla 65535, ovvero, in numeri esadecimali, dalla 4000 alla FFFF).

P_RAMT
UDG
RAMTOP
65535 (48K)
32767 (16K)
Grafica definita dall'utente
Programma
stack
file
etc.,
RAM
23734
23733
Variabili di sistema
23552
23551
Buffer di stampante
23296
23295
Attributi
22528
22527
File di display
16384
16383
ROM
Ø

40

L'insieme di 16K di ROM e 48K di RAM dà un totale di 64K di memoria, cioè 65536 byte.

Prima di andare oltre, faremmo bene ad eliminare ogni residuo di dubbio o confusione riguardo a questo numero "65535". Ormai dovrebbe essere chiaro che la CPU dello Spectrum può manipolare 256 byte diversi di 8 cifre ciascuno. Immaginiamo adesso, per ipotesi, che la sua memoria sia come un enorme edificio con tante stanze, alto 256 piani e con 256 stanze diverse per ogni piano. A ciascuna "stanza" può essere assegnato un indirizzo formato dal numero del piano su cui essa si trova e dalla posizione su quel piano. Supponendo che i piani siano numerati in sequenza dallo 0 al 255 su ciascuno dei 256 piani, diventa possibile individuare ogni singola stanza specificando i suoi numeri di "piano" e di "porta". Per esempio: piano numero 10, porta 54. L'individuazione poi di quell'indirizzo nel totale di 65536 stanze può essere facilmente determinato come segue:

(Numero di piano × 256) + Numero di porta.

$$(10 \times 256) + 54 = 2614$$

In modo del tutto analogo, l'indirizzo più alto sarebbe:

$$(255 \times 256) + 255 = 65535$$

Questo è in pratica il modo in cui lo Spectrum riesce ad indirizzare le sue 65536 locazioni di memoria, sebbene però, per via del tipo di funzionamento dello Z80, esso lo faccia "alla rovescia": prima il numero di porta, poi quello del piano. Due byte di memoria vengono utilizzati per comporre un indirizzo: il primo contiene la parte "meno significativa" dell'indirizzo (54 nell'esempio di prima), il secondo contiene quella "più significativa" (10 nell'esempio). I codici di indirizzo usati dalla CPU Z80 sono quindi due byte da 8 bit ciascuno, considerati come un singolo numero di 16 cifre: di queste, le prime otto sono chiamate il "Byte meno significativo" (in inglese "Least Significant Byte", abbreviato in LSB) , e le altre otto il "Byte più significativo" ("Most Significant Byte", che si abbrevia in MSB).

Tutto ciò può essere esemplificato (ed utilizzato per uno scopo molto interessante) mediante la seguente formuletta per determinare quanti byte di memoria sono stati utilizzati per memorizzare un programma in Basic. Caricate un qualsiasi programma in Basic, poi trovate quanti byte sono stati utilizzati in questo modo:

```
PRINT (PEEK 23653+(PEEK 23654*256)
-(PEEK 23635+PEEK 23636)*256)
```

Questa istruzione identifica semplicemente l'indirizzo in cui inizia la memoria libera (STKEND) e gli sottrae l'indirizzo di inizio del programma Basic. Questi indirizzi vengono determinati utilizzando il comando PEEK, cioè leggendo le

"variabili di sistema", e ricordando che ogni indirizzo è memorizzato in due byte di memoria di cui il primo è il meno significativo. Il primo indirizzo allora è stato trovato "guardando dentro" la locazione di memoria numero 23653, estraendo il suo contenuto (un byte di otto cifre) e sommandogli il contenuto della successiva locazione di memoria moltiplicato per 256. Analogamente per il secondo indirizzo.

Discutendo degli indirizzi ho anche accennato al comando PEEK. Questa parola inglese significa guardare, sbirciare, e descrive bene la funzione svolta da questo comando: esso ci consente di guardare dentro una locazione di memoria e vedere il suo contenuto, proprio come se guardassimo attraverso la finestra di una delle 65536 stanze del nostro edificio. Nella parte di memoria assegnata alla ROM possiamo fare soltanto questo: guardare senza toccare! nella parte RAM della memoria, invece, possiamo sia guardare che toccare. Il comando per "toccare", ovvero cambiare il contenuto di una locazione di memoria è "POKE" (che significa appunto frugare); esso consente all'utente di modificare il contenuto di una locazione di memoria, inserendovi un nuovo valore. Provate! Chiedete al vostro Spectrum di eseguire: "PRINT PEEK n", dove "n" è il numero di una locazione RAM a vostra scelta. Dopo aver visto comparire sul video il contenuto di quella locazione, fate: "POKE n,x", dove "n" è la stessa locazione di prima, e "x" è un qualsiasi numero nell'intervallo da 0 a 255. Adesso fate di nuovo "PRINT PEEK n" e verificate che avete effettivamente cambiato il contenuto di quella locazione di memoria. Nel seguito di questo libro farete veramente moltissima pratica con i comandi PEEK e POKE!

Forse senza rendercene veramente conto, siamo giunti al punto di essere quasi del tutto pronti per iniziare a parlare allo Spectrum nella sua lingua. Sappiamo come sono fatte le "parole", dove sono immagazzinate e come, in teoria, le immettiamo nel computer (tramite il comando POKE). Che altro abbiamo bisogno di conoscere? Chiaramente dobbiamo essere in grado di preparare il computer a ricevere le nostre istruzioni, e poi sarà necessario assicurarsi di quali sono precisamente le parole che il computer può capire ed elaborare. In senso stretto, il micrprocessore Z80 dovrebbe essere in grado di capire soltanto 256 parole di linguaggio macchina, ma anche qui, senza che ci si debba sorprendere troppo, esso riesce ad espandere abbastanza le proprie possibilità, e così, mediante un opportuno uso di "prefissi" supplementari, oltre 600 parole possono essere riconosciute dalla CPU. Esse sono tutte elencate nell'appendice B. Vediamo ora alcuni concetti elementari.

Le routine in linguaggio macchina possono sempre essere chiamate in azione in un programma BASIC mediante il comando "RAND USR". Perciò, per accedere ad una routine in linguagio macchina si usa una "User Subroutine", specificando il luogo della memoria in cui tale routine può essere trovata. Per esempio, "RAND USR 30000" permette di richiamare una routine scritta in linguaggio macchina, il cui primo byte sia stato memorizzato alla locazione numero 30000; il seguito

della routine è memorizzato in sequenza da quel punto in su. Come si fa a determinare inizialmente la locazione di inizio di una routine? È semplice: basta che noi scegliamo un punto qualsiasi e poi lo ordiniamo al computer. Inserendo il comando "CLEAR 29999" (oppure qualsiasi altra locazione conveniente) otteniamo l'effetto di fissare il "RAMTOP" a quell'indirizzo, cosicché esso viene identificato come l'ultima locazione disponibile per memorizzare una qualsiasi parte di un programma BASIC. La routine in linguaggio macchina inizierà allora alla locazione successiva: 30000. Il comando "NEW" cancella il contenuto della memoria RAM solo fino al punto RAMTOP, e quindi qualsiasi routine in linguaggio macchina scritta al di sopra del RAMTOP risulta protetta sia contro la cancellazione con "NEW", sia contro la scrittura di un programma BASIC sopra di essa. Il comando RUN, a sua volta ripulisce e reinizializza tutte le variabili, senza però cambiare in alcun modo il punto RAMTOP. Di conseguenza, l'uso del comando "CLEAR" con uno specifico indirizzo di memoria è un modo per spostare il punto RAMTOP verso l'alto, in modo da avere più spazio per un programma BASIC (utilizzando cioè parte dello spazio dedicto ai simboli grafici definiti dall'utente), oppure verso il basso, per aumentare l'area di memoria RAM che è protetta nei confronti del comando "NEW".

Adesso che abbiamo deciso si deve memorizzare il linguaggio macchina, ci chiediamo: come si fa per memorizzarlo? Abbiamo già accennato che esso deve essere caricato in modo sequenziale, poiché questo è il modo in cui funziona lo Spectrum. Il processo di ricerca ed esecuzione del programma, con i comandi "FIND" ed "EXECUTE", ha inizio all'indirizzo specificato e poi continua con tutte le successive locazioni della memoria, interpretando tutto ciò che viene trovato come istruzione ed eseguendola. Questo processo continua finché si arriva all'istruzione di arresto, oppure fino a quando il computer "impazzisce". Di seguito viene riportato un divertente programma che mostra tale funzionamento "sequenziale" del computer. Per favore non preoccupatevi di come funziona: esso si basa soprattutto sul fatto che la ROM dello Spectrum è diventata del tutto farneticante.

```
  5 RESTORE
 10 READ n
 20 LET a=120
 25 PLOT 55,27: DRAW a,a,n*PI
 32 CLS
 33 IF n=9999 THEN GO TO 5
 35 GO TO 10
 40 STOP
 50 DATA 597,631,315,343,297,63
1,613,787,313,741,187,147,279,99
99
```

Dopo questa digressione, vediamo quatro possibilità per il caricamento di istruzioni in linguaggio macchina:

1. inserimento di un byte alla volta con il comando "POKE": POKE x,y

dove x è l'indirizzo iniziale e y è il numero da caricare. Il byte successivo viene caricato nell'indirizzo x+1, poi nell'x+2, e così via.

2. anelli "FOR/NEXT".

```
10 FOR A = 1 TO X
20 INPUT Y
30 POKE A +    loc. iniziale    Y
40 NEXT A
```

3. Lettura di dati con "READ".

```
10 FOR A = 1 TO X
20 READ N
30 POKE loc. iniziale + A,N
40 NEXT A
50 DATA N1,N2,N3,N4...
```

dove N1,N2,N3,N4... sono i byte in successione della routine in linguaggio macchina.

4. Caricamento da nastro.

```
CLEAR loc. scelta
LOAD "nome" CODE
```

I metodi 2 e 3 costituiscono due basi diverse per un programma di caricamento di linguaggio macchina. Chiaramente il caricamento di numero binari è una seccatura; possiamo invece usare bene le nostre nuove conoscenze sul sistema esadecimale, e, voilà! ecco un programma "HEX.LOADER":

```
  10 PRINT "         ***HEX LOADER
***"
  20 PRINT "         © James Walsh
 1983"
  30 INPUT ''"Indirizzo di inizi
o?";ind
```

```
  40 PRINT '''"Introduci i codic
i uno per volta"
  50 PRINT "usando solo le MAIUS
COLE"
  60 PRINT ind;" = ";
  70 INPUT a$
  80 IF a$="FINE" THEN GO TO 100
0
  85 IF LEN a$<>2 THEN GO TO 70
  90 LET hi=CODE a$(1)-48
 100 IF hi>9 THEN LET hi=hi-7
 110 LET ans=16*hi
 120 LET low=CODE a$(2)-48
 130 IF low>9 THEN LET low=low-7
 140 LET ans=ans+low
 150 PRINT a$
 160 POKE ind,ans
 170 LET ind=ind+1
 180 GO TO 60
1000 PRINT AT 21,0;"Programma te
rminato"
```

CAPITOLO 4

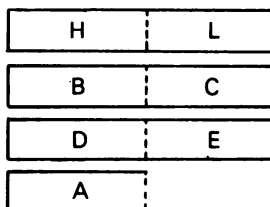
UTILIZZO DEI REGISTRI

Dopo aver letto i capitoli precedenti, dovreste esservi fatta un'idea abbastanza precisa dell'insieme di numeri che il computer può capire, del modo in cui noi possiamo usare questi numeri e del modo in cui il computer li utilizza per permetterci di manipolare la memoria del computer stesso. Ciò avviene proprio come se ci fosse una lunga fila di scatole vuote, ciascuna identificata da uno specifico numero cosicché possiamo accedervi direttamente, senza dover prima guardare in quelle che la precedono; possiamo poi aggiungere o togliere dati in queste scatole in qualsiasi momento. Ciò che faremo in questo capitolo è vedere come il computer usi questi numeri quando sono fuori da quelle scatole, e come noi possiamo manipolarli. Vedremo come il computer memorizzi questi numeri e i vari comandi in linguaggio macchina. Cominceremo poi ad esplorare i modo in cui queste istruzioni ci possano essere utili nei nostri programmi. Alla fine vi sarà dato un problemino da risolvere con l'uso del linguaggio macchina.

I registri

È già una buona cosa essere in grado di conservare dei numeri in certe scatole e poi di andarli a riprendere, ma in pratica ciò non è di grande utilità se poi non si riesce a fare niente di più. Quando si scrive un programma in BASIC si usano le variabili. Per esempio, se si vuole che un numero sia memorizzato come variabile A, si userà l'istruzione "LET A = il numero". Dopodiché si sarà in grado di compiere qualsiasi operazione con tale variabile A: la si potrà sommare ad un altro numero, raddoppiarla, sottrarla da qualche altro numero, e quando si è finito si può azzerare di nuovo quella variabile facendo semplicemente: "LET A = 0".

Sfortunatamente non esistono variabili in linguaggio macchina. Invece ci sono i cosiddetti "registri". Ci sono sette registri da un solo byte ciascuno che possono essere usati e manipolati molto facilmente. Ne esistono altri, ma questi sono utilizzati prevalentemente dal computer: li possiamo usare anche noi, sebbene con maggior difficoltà, e per questo motivo per il momento li lasciamo da parte. Per la grande maggioranza del tempo che dedicherete alla programmazione in

linguaggio macchina i registri che userete di più saranno questi sette di "facile uso". Ciascun registro è identificato da una lettera: A,B,C,D,E,H,L. Non c'è un vero motivo al perché della scelta di queste lettere in particolare, quindi non preoccupatevi di trovare un senso in questa convenzione. Come già sapete, considerando un solo byte il computer può usare solo un numero compreso fra 0 e 255. Poiché ciascuno di questi registri è un "registro ad un solo byte", 255 è il massimo numero immagazzinabile in un registro. Senza dubbio avrete già notato che questo è veramente un numero troppo basso per ogni utilizzo pratico: quante volte riuscite effettivamente a scrivere un programma senza numeri maggiori di 255? Per rimediare a ciò il computer raggruppa insieme due registri da un solo byte per formare la cosiddetta "coppia di registri". Se si combinano due registri il massimo numero che può essere raggiunto sarà 256 × 256, meno 1, cioè un totale di 65535. Nella terminologia dei computer questo intervallo da 0 a 65535 viene indicato con 64K. Il motivo per cui dobbiamo sottrarre uno nel conto precedente è che l'intervallo parte da zero, per cui si deve aggiungere un altro byte per registrare il numero zero. I registri non possono essere accoppiati in modo qualsiasi. Le coppie fisse sono: BC, DE, HL, come si può ben vedere nel disegno che segue.

Se adesso date una rapida occhiata alla lista dei comandi in linguaggio macchina e dei loro codici esadecimali alla fine di questo libro, noterete che nella colonna intitolata "mnemoniche" c'è un gran numero di comandi che fa riferimento direttamente a questi registri. Vi renderete conto che una pienpa comprensione della natura e del funzionamento di tali registri sarà di un'utilità maggiore di qualsiasi altra cosa nella programmazione in linguaggio macchina. Un'altra possibilità della CPU Z80, che sarà anch'essa di notevole utilità, è la sua capacità di usare i registri come registri ad un solo byte, cioè con un'estensione da 0 a 255 se presi singolarmente, oppure da 0 a 65535 se a coppie. Questa è una delle principali caratteristiche di versatilità del computer.
Fra poco vedremo come caricare dei numeri in questi registri e come eseguire somme e sottrazioni. Successivamente vedremo come effettuare manipolazioni più complicate di questi e di altri registri. Sebbene gli altri registri siano per noi di minore utilità, ed alcuni siano in effetti riservati per uso proprio del computer, ci saranno occasioni in cui riuscirà molto vantaggioso accedere a quesi registri dai nostri programmi.

Probabilmente il modo più semplice per affrontare il concetto di "registro" è quello di immaginarsi il computer come una persona che indossa una giacca. Questa giacca ha sette tasche, ciascuna tasca un'etichetta particolare. Il computer può facilmente estrarre degli oggetti da queste tasche e inserirvi altri oggetti. Per fare ciò è necessario dare degli "ordini" al computer, dato che esso non ha la capacità di agire autonomamente; perciò le istruzioni devono essere molto precise. Il computer può capire soltanto i comandi che gli vengono dati sotto forma di numeri o di codici. Il linguaggio con il quale i comandi sono espressi unicamente come numeri viene chiamato "linguaggio macchina". Questo è in pratica il fondamento del linguaggio macchina. Purtroppo, mentre è abbastanza semplice inserire i comandi nella memoria del computer in forma di numeri, usando l'istruzione "POKE", non è così semplice per noi poveri mortali cercare di ricordare il codice relativo ad ogni comando. Non disperate comunque, poiché essendo questo un problema di vecchia data, esiste già una facile soluzione. La risposta al nostro problema è fornita dal linguaggio "Assembly". Esso consiste in un insieme di istruzioni che corrispondono esattamente ai codici del linguaggio macchina, ma che sono molto più comprensibili per noi. Per esempio, un'istruzione in linguaggio macchina potrebbe essere questa: 01101011, mentre il corrispettivo in assembler è "LD A, 1".

Sebbene ciò possa ancora sembrarci quasi privo di significato, ciò che dovrebbe risultare immediatamente chiaro è che la nostra normale capacità di interpretare il significato di "gruppi" di lettere o di numeri di frequente utilizzo ci consente di lavorare con maggiore speditezza con le istruzioni "assembly", piuttosto che con i loro corrispondenti codici binari. Ciò diverrà sicuramente ancora più chiaro quando si cercherà di imparare a memoria alcuni o molti di questi codici. Queste rappresentazioni "umane" dei comandi binari vengono di solito chiamate "mnemoniche". Purtroppo però non possiamo inserire queste mnemoniche direttamente nel computer, semplicemente perché il computer non capisce le nostre mnemoniche.

Dobbiamo quindi convertirle ad una forma che sia comprensibile al nostro povero computer. Ciò vuol dire che in un modo o nell'altro dobbiamo tornare al codice binario. Ci sono due metodi fondamentali per ottenere questo risultato. Un modo molto lento ma efficace è semplicemente quello di usare le tabelle di conversione alla fine del libro. L'altro modo più semplice, e quindi molto più rapido è quello di usare un programma "assembler" già pronto.

Nel seguito del libro cercherà di presentare tutti i principali comandi insieme alle loro mnemoniche. Ne spiegherò il significato e l'uso, e nello studio degli esempi che verranno dati nel libro effettuerete voi stessi la conversione da mnemoniche a codice operativo, caricando poi i vari programmi con il programma "LOADER" già visto. Chiaramente un assembler vi sarebbe di notevole aiuto, ma non c'è bisogno che voi adesso usciate di corsa per andare subito a comprarlo. Se

decidrete di voler risparmiare fatica o comunque di voler acquistare un assembler per averlo a disposizione quando me ne occuperò più dettagliatamente, allora potete trovare la lista dei fornitori dell'assembler da me usato alla fine di questo volume. Può essere una buona idea almeno farselo ordinare.

È importante ricordare che il linguaggio assembly non è un adattamento del linguaggio macchina, come il Basic. Nell'assembly c'è una sola mnemonica per ogni istruzione in linguaggio macchina, e viceversa c'è un'unica istruzione in linguaggio macchina per ogni mnemonica in assembly. Possiamo quindi dire che l'assembler è in pratica equivalente al linguaggio macchina. Quasi tutte le mnemoniche sono abbreviazioni delle operazioni eseguite da una particolare istruzione in linguaggio macchina. Si tratta quindi, quasi sempre, di riconvertire tale abbreviazione all'istruzione corrispondente. Per esempio: "INC HL" è l'abbreviazione di "incrementa HL"; analogamente "LD A,0" è l'abbreviazione di "load A con 0". I codici in linguaggio macchina corrispondenti a queste istruzioni sono rispettivamente "23" e "62,0".

Ogni persona può scegliere di listare i propri programmi in vari modi, cioè in linguaggio macchina o in linguaggio assembly. È molto più facile capire un programma scritto in assembly, ma un assembler per il vostro Spectrum è piuttosto costoso, intorno alle venti, trentamila lire. Se però si mette a confronto questa spesa con il tempo e la fatica che potete risparmiare nel caricare i programmi, oltre alla considerazione che, essendo in grado di capire quanto è scritto, potrete imparare, allora l'assembler è veramente un aiuto prezioso. Se comunque decidete di non usare un assembler, o non ne avete uno per un motivo qualsiasi, sarà lo stesso abbastanza facile caricare i programmi nella macchina tramite un "hex loader". Per rendervelo possibile le liste di tutti i programmi di questo libro sono comprensive sia del linguaggio assembly che del linguaggio macchina, di modo che, qualunque sia il modo in cui deciderete di inserire i programmi, potrete sempre guardare la versione in assembly per capire come funzionano.

Ricordate che, essendo il linguaggio macchina basato direttamente sulla CPU, nel nostro caso sullo Z80, se siete in grado di scrivere un programma nel linguaggio macchina dello Spectrum, non ci vorranno grossi sforzi per riuscire a scrivere nel linguaggio macchina dello ZX81, dello ZX80 e dello Spectrum Plus, oppure chiaramente, di qualsiasi altro computer che usi la CPU Z80.
Ciò può essere particolarmente importante da ricordare quando si deve decidere quale computer acquistare dopo lo Spectrum, oppure se vorrete comprarvi un'altro micro! Per esempio, se vi comprerete un BBC, oppure un Commodore VIC20, dovrete impararvi di nuovo un bel po' del linguaggio macchina prima di poterlo usare su quei computer, mentre per lo ZX81 ci sarebbero poche cose da capire prima di poter scrivere dei buoni programmi in linguaggio macchina. Questa non è certo l'unica considerazione da fare per decidere se si deve passare o no ad

un altro computer, ma è un fattore che può essere utilmente ricordato.
Bene, adesso facciamo un intervallo. Ecco un interessante programma che funge da esempio su che cosa è veramente il linguaggio macchina. Notate le due sezioni del programma: la lista in assembler e quella in linguaggio macchina. Questo programma fa muovere verso sinistra i pixel (i singoli punti che compongono l'immagine sul video).

```
org 30000
30000 21 FF 57      ld hl,22527
30003 E5            push hl
30004 11 20 5B      ld de,23328
30007 01 20 00      ld bc,32
30010 ED B8         lddr
30012 E1            pop hl
'HL=IND BASSO DEL BLOCCO BASSO
'B=NUM. di blocchi
30013 06 03         ld b,3
A
30015 C5            push bc
30016 06 08         ld b,8
B
30018 C5            push bc
30019 E5            push hl
30020 06 08         ld b,8
C
30022 C5            push bc
30023 E5            push hl

30024 E5            push hl
30025 D1            pop de
30026 25            dec h
30027 01 20 00      ld bc,32
30030 ED B8         lddr
30032 E1            pop hl
30033 25            dec h
30034 C1            pop bc
30035 10 F1         djnz C
30037 24            inc h
30038 11 E0 06      ld de,1760
30041 E5            push hl
30042 19            add hl,de
30043 D1            pop de
30044 01 20 00      ld bc,32
30047 ED B8         lddr
30049 E1            pop hl
30050 11 20 00      ld de,32
30053 ED 52         sbc hl,de
30055 C1            pop bc

30056 10 D8         djnz B
30058 11 00 07      ld de,1792
30061 ED 52         sbc hl,de
30063 E5            push hl
30064 11 20 00      ld de,32
30067 19            add hl,de
```

```
30068 E5            push hl
30069 D1            pop de
30070 E1            pop hl
30071 E5            push hl

30072 01 20 00      ld bc,32
30075 ED B8         lddr
30077 E1            pop hl
30078 C1            pop bc
30079 10 BE         djnz A
30081 11 20 00      ld de,32
30084 19            add hl,de
30085 EB            ex de,hl
30086 21 20 5B      ld hl,23328
30089 06 20         ld b,32

D
30091 7E            ld a,(hl)
30092 00            nop
'ld a,0 per non ritorno
30093 12            ld (de),a
30094 1B            dec de
30095 2B            dec hl
30096 10 F9         djnz D
30098 C9            ret
```

Semplici comandi sui registri

Avete già preso familiarità con il modo in cui i registri ad un solo byte possono combinarsi in coppie di registri. È bene adesso rammentarvi che i registri singoli possono tenere numeri da 0 a 255, mentre le coppie di registri da 0 arrivano a 65535. Cominciamo con il vedere come caricare numeri nei registri singoli.

Supponiamo di voler inserire un numero, per esempio 11, in uno dei registri (o "tasche"). Per fare ciò dobbiamo "caricare" quel numero nella tasca, usando il semplice comando dello "LD", l'abbreviazione di "load". Poiché ci sono diverse "tasche" o registri, bisogna che si sia un ugual numero di codici in linguaggio macchina per ognuna di questa istruzioni. I codici per queste mnemoniche di caricamento sono i seguenti:

OP CODE	HEX	DECIMAL
LD A,xx	3Exx	62,xx
LD B,xx	Ø6xx	6,xx
LD C,xx	ØExx	14,xx
LD D,xx	16xx	22,xx
LD E,xx	1Exx	3Ø,xx
LD H,xx	26xx	38,xx
LD L,xx	2Exx	46,xx

Prima si deve identificare il registro in cui si vuol fare il caricamento; poi si deve indicare il numero da caricare nel registro prescelto. Ciò vuol dire identificare A, B, C, o un altro registro e poi specificare il numero da inserire. La rappresentazione in assembler per caricare il valore 11 nel registro A è la seguente: "LD A,0B". Però prima di poter far eseguire questa istruzione bisogna usare un programma assembler in modo che il computer possa capire che cosa gli si chiede di fare. Se invece si usa un programma "hex-loader" (cosa che io suggerirei di fare per il momento, così da capire più facilmente cosa sta succedendo), allora si deve prima inserire il numero 3E: ciò indica al computer che deve effettuare un'operazione di caricamento nel registro A. Questo deve essere seguito dal numero 0B, ed il computer capirà che questo è il numero da caricare nel registro A.

NOTA: È importante ricordare che quando viene fatto riferimento ad un codice operativo oppure ad un comando in assembler indicando due lettere "x", quattro lettere "x" oppure le lettere "d" dopo il comando stesso, ciò significa che in quel punto si possono inserire uno o due byte a propria scelta. Per esempio, nell'indicazione del comando "LD A,XX" le due x stanno a significare che dobbiamo specificare un numero da un solo byte dopo il comando "LD A", da caricare nel registro A.

Con la stessa facilità possiamo caricare un numero in qualsiasi altro registro: A, B, C, D, E... usando questo metodo, ma con un codice o mnemonica assembler diverso per ogni registro (vedi sopra). Prima di andare avanti è meglio vedere bene come funziona tutto ciò. Ci sono due cose importanti da ricordare (anche se non è del tutto cruciale capire perfettamente questo punto), prima che possiamo procedere a scrivere ed eseguire un programma in linguaggio macchina. La prima cosa è che alla fine di un programma in linguaggio macchina è necessario inserire il codice di ritorno ("RET", abbreviazione di "return"). "RET" funziona esattamente come "RETURN" in una subroutine Basic. Per esempio, se si inserisce l'istruzione "GOSUB 1000" in un programma Basic, alla fine della subroutine che inizia alla riga 1000 si inserirà l'istruzione "RETURN", per indicare al computer che si desidera ritornare all'istruzione subito dopo la riga "GOSUB 1000". Allo stesso modo, se dal Basic entriamo in una routine in linguaggio macchina, usando il comando "USR" del Basic che significa "User SubRoutine", sarà poi necessario che inseriamo il comando "RET" alla fine della routine in linguaggio macchina, in modo da ritornare al programma Basic e proseguire l'esecuzione con l'istruzione successiva. Fra poco vedremo perché si deve fare così, ma per il momento è importante solo che vi rendiate conto di tale necessità. Il codice in linguaggio macchina per il comando "RET" è "C9". L'altra cosa da ricordare è che se si entra in un programma in linguaggio macchina tramite il comando "PRINT USR XXXXXX", il numero che verrà stampato quando termina l'esecuzione della user subroutine è il contenuto della coppia di registri BC. Possiamo ora usare questo fatto con molto profitto.

Il programma nella figura che segue non fa altro che caricare 0 nei registri B e C, e poi fa ritorno al Basic stampando 0. Per caricare questo programma usate il programma del caricamento del codice di macchina, oppure il programma "hex-loader" già visto, inserendo semplicemente i numeri esadecimali indicati in figura. Quando il programma è stato caricato nella memoria del computer, non cercate di farlo eseguire con l'usuale comando "RUN".
Scrivete invece "STOP" e fate eseguire il comando "PRINT USR 30000" direttamente dal Basic. In questo modo la routine in linguaggio macchina farà ritorno direttamente al modo di comando, in Basic, stampando contemporaneamente il contenuto di BC.
Dopo aver fatto ciù provate a modificare il programma inserendo diversi numeri nei registri B e C, e cercate di capire come verrà stampato il loro contenuto. Ricordate che state usando un numero a due byte, con un massimo di 255 in ciascun byte.

OP CODE	HEX
LD B,ØØ	Ø6ØØ
LD C,ØØ	ØEØØ
RET	C9

La progressione naturale è quella di procedere dal semplice caricamento di numeri nei vari registri all'analogo problema di caricare il contenuto di un registro in un altro registro. Dobbiamo usare un diverso codice di linguaggio macchina per ciascuna di queste operazioni (per esempio per caricare il registro B nel registro A), il che risulta essere un gran numero di codici diversi; ci concentreremo perciò a questo punto sul caricamento di diversi registri nel registro A, per semplificare la spiegazione e la comprensione. Per tutti gli altri registri il principio è esattamente lo stesso, tranne per il fatto che cambiano i codici in linguaggio macchina e le mnemoniche. Quindi, dopo averne capito uno li si è capiti tutti.

Per caricare il contenuto del registro B nel registro A, si può usare il semplice comando "LD A,B" ("con il significato di carica in A il contenuto di B"). Se torniamo al nostro iniziale paragone tra il computer e una persona in giacca, possiamo pensare che questa operazione sia costituita dalle seguente fasi:

1. guardare nella tasca A, tirare fuori il suo contenuto e buttarlo via;

2. guardare nella tasca B, contare quanti oggetti vi sono contenuti senza però rimuoverli;

3. raccogliere tanti oggetti quanti ce n'erano in B e metterli in A.

A questo punto c'è un uguale numero di oggetti nella tasca A e nella B. È importante notare che il valore iniziale del contenuto di A non ha alcuna rilevanza sul risultato finale dell'operazione.

Nel registro A possono essere caricati i valori contenuti in qualunque altro registro.Le mnemoniche e i codici esadecimali in linguaggio macchina per queste istruzioni sono indicati di seguito:

OP CODE	HEX	DECIMAL
LD A,A	7F	127
LD A,B	78	120
LD A,C	79	121
LD A,D	7A	122
LD A,E	7B	123
LD A,H	7C	124
LD A,L	7D	125

Il passo successivo da fare è quello di guardare quali sono le mnemoniche e i codici in linguaggio macchina per le istruzioni che caricano il contenuto di altri registri nei registri BC, poi usare questi registri, cosicché il risultato dell'operazione venga mostrato sul video al momento del ritorno al Basic.

CAPITOLO 5

LE OPERAZIONI CON I REGISTRI

Adesso che sappiamo cosa sono i registri e come possiamo immettervi dei numeri, caricandoli direttamente oppure caricando in un registro il contenuto di un altro registro, dobbiamo passare a pensare in che modo possiamo manipolarli. Se avete avuto modo di scrivere programmi in Basic saprete certamente che è inutile essere capaci di assegnare un valore ad una variabile se poi non si riesce a modificarlo per via aritmetica. In altre parole, per poter "lavorare" con un numero è necessario poter eseguire con esso tutte le varie operazioni aritmetiche. Questo è quando vedremo in questo capitolo. Vedremo anche cosa succede quando un numero è troppo grande o troppo piccolo per poter essere contenuto in un registro.
A questo proposito potremmo studiare il segnale di riporto detto "carry flag" e le altre informazioni collegate che sono registrate nel computer.

Ormai avrete già cominciato a pensare al modo in cui riusciremo a fare la somma di due registri, e se riusciremo o meno a farlo nello stesso modo in cui si opera in Basic, dove per sommare il valore della variabile XY a quella della variabile AB si scrive semplicemente "LET AB = AB+XY". Il risultato verrà poi memorizzato in AB. Per nostra fortuna è possibile eseguire questa operazione anche in linguaggio macchina in modo molto più semplice anche se l'istruzione stessa è del tutto diversa. Se desiderassimo sommare il contenuto della coppia di registri De a quello dei registri HL, invece di scrivere "LET HL = HL+DE" (come si farebbe in Basic), è necessario usare il comando "ADD HL,DE". Il seguente esempio mostra chiaramente cosa succede se usiamo questa istruzione in un programma. Vale la pena notare che le prime due righe contengono i numeri da sommare tra loro; la terza riga è l'istruzione che svolge effettivamente l'operazione, e le righe 4 e 5 memorizzano il risultato nel registro BC.
Quest'ultimo passaggio deve essere fatto usando due comandi separati, dato che non esiste un'istruzione del tipo "LD BC,HL". Infine, l'ultima istruzione, "RET", riporta semplicemente il computer al Basic. Vi raccomanderei di provare subito questo programma.

```
ORG 30000
30000 21  01  00   LD HL,0001
30003 11  00  80   LD DE,32676
30006 19           ADD HL,DE
30007 44           LD B,H
30008 4D           LD C,L
30009 C9           RET
```

È per noi un vantaggio il fatto che il massimo numero che può essere memorizzato in una coppia di registri sia oltre 65.000, di modo che solitamente si può essere abbastanza sicuri che il risultato della somma di due numeri non andrà oltre questo limite. Comunque è utile sapere che cosa accadrebbe se ciò si verificasse: il breve programma qui sotto dovrebbe chiarire questo punto. Provatelo e trovate la risposta.

```
org 30000
30000 21 01 00     ld hl,0001
30003 11 FF FF     ld de,65535
30006 19           add hl,de
30007 44           ld b,h
30008 4D           ld c,l
30009 C9           ret
```

Prima di dire la risposta, vediamo velocemente insieme come funziona quest'ultimo esempio. La prima istruzione memorizza il numero 1 nella coppia di registri HL. Sì, direte voi, però il numero che segue il "21" nel codice in linguaggio macchina non è 1, ma "0100". Eccoci giunti ad incontrare una caratteristica un poco strana del linguaggio macchina che è bene vi sia chiarita prima di procedere oltre. Quando carichiamo un numero in una coppia di registri, invece di scrivere prima il byte più elevato, cioè quello più significativo, e poi quello più basso, facciamo il contrario. Quindi, per memorizzare 1 nel registro HL bisogna scrivere prima 01, poi 00, cioè "0100". Se non vi è molto chiaro cosa si intende per "byte più significativo", sarà bene che ritorniate a leggere la sezione più indietro in questo libro, nella quale questi concetti venivano spiegati.
Giusto per chiarirvi le idee eccovi alcuni esempi veloci di brevi routine che fanno uso di questo metodo. Comunque, il caricamento del byte più significativo dopo quello meno significativo non è strano e dovrebbe presto riuscirvi spontaneo. Adesso provate queste brevi routine.

1.

```
org 30000
30000 01 00 01     ld bc,0100
30003 C9           ret
```

Il risultato che vogliamo è 1, però in questo esempio abbiamo caricato il numero sbagliato: per questo motivo il risultato è 256. Quando si scrive in assembler è consuetudine scrivere i numeri in esadecimale e nel giusto ordine di byte, cioè prima il byte più significativo, poi quello meno significativo. Come potete vedere però, in questo primo esempio si è sbagliato nel fare la conversione al codice esadecimale, e il primo byte è risultato quello più significativo. Questo primo esempio è dunque errato.

2.

```
org 30000
30000 01 01 00      ld bc,0001
30003 C9            ret
```

Questa volta il byte più significativo è stato scritto dopo quello meno significativo nel codice esadecimale, e quindi abbiamo la risposta corretta: 1.

3.

```
org 30000
30000 01 FA 7B      ld bc,64123
30003 C9            ret
```

Di nuovo in questo esempio vediamo che i due byte più elevato e meno elevato sono stati scritti nell'ordine sbagliato: la risposta è quindi del tutto sbagliata. Ora, prima di fare eseguire la routine dell'esempio successivo, il numero 4, provate a calcolarvi in decimale quale dovrà essere il risultato esatto.

4.

```
org 30000
30000 01 7B FA      ld bc,64123
30003 C9            ret
```

Adesso, prima di tornare al primo esempio che stavamo considerando, possiamo dire in sintesi che, quando si deve caricare un numero in una coppia di registri, il byte più significativo va scritto dopo il byte meno significativo nella codifica esadecimale del linguaggio macchina; in assembler, invece, si conserva l'ordine consueto dei due byte.

Questo è un altro esempio dell'aiuto che ci può dare un assembler, dato che ci consente di scrivere i numeri con la normale configurazione: il byte più significativo prima di quello meno significativo.

Ritorniamo ora all'esempio di prima.
Abbiamo già chiarito che la prima istruzione memorizza 1 nella coppia di registri HL. L'istruzione successiva memorizza il numero esadecimale FFFF nei registri DE; vi ricordate che questo numero corrisponde al decimale 65535, e che rappresenta il numero più elevato che può essere memorizzato in una coppia di registri. Se adesso sommiamo a questo numero il numero 1 otterremo una condizione di "overflow" (fuori scala). Quando in BASIC succede ciò il computer risponde con un messaggio di errore arrestando l'esecuzione del programma. In linguaggio macchina questo non succede, poiché non ci sono codici di errore, però l'esecuzione risulta errata. In questo caso particolare succede semplicemente che il computer, invece di arrivare alla soluzione corretta 65536, che esce dai suoi limiti, torna a zero, ricordando questo evento con un "segnale di riporto" (carry flag) posto uguale a 1. Non vi preoccupate per il momento di capire cos'è il segnale di riporto, dato che ci ritorneremo sopra fra non molto.

Ci sono due cose importanti da ricordare riguardo le addizioni di due coppie di registri. Anzitutto, un coppia di registri può essere sommata soltanto alla coppia HL. Poi, coppie di registri possono essere sommate soltanto ad un'altra coppia di registri: un registro da un solo byte non può essere sommato ad una coppia di registri, e viceversa. Di seguito viene riportata una liste dei codici esadecimali per effettuare la somma di una coppia di registri con un'altra coppia di registri. Vi potrà sembrare strano che esista anche un comando per sommare HL a se stesso, ma in effetti questo è un modo molto semplice per raddoppiare il contenuto in HL. Si noti pure che questi comandi richiedono un solo byte: confrontando questi comandi con le equivalenti istruzioni in BASIC ci si rende subito conto della notevole differenza, dato che il BASIC richiede circa 10 volte più memoria del linguaggio macchina!!!

```
org 30000
30000 09              add hl,bc
30001 19              add hl,de
30002 29              add hl,hl
```

Vediamo adesso come si sommano assieme registri singoli. Ricordando che i registri singoli possono contenere numeri compresi fra 0 e 255, la possibilità di uscire da tali limiti, cioè di un "overflow", è molto più frequente che con le coppie di registri. Cosa succede quando si va in "overflow"? Proprio come con le coppie di registri: si torna a zero. Il problema è però il seguente: cosa possiamo fare quando si verifica questa circostanza? In effetti c'è qualcosa che possiamo fare. Ogni volta che sommiamo due numeri si possono avere solo due casi: o si esce dai limiti e c'è un "overflow" (o riporto) oppure ciò non succede. Il computer stesso dedica un registro molto particolare proprio per il controllo di queste situazioni. Esso viene chiamato il registro "F". Noi di solito non useremo mai il

registro F, proprio perché esso viene monopolizzato dal computer per memorizzare varie piccole informazioni. Ciò è reso possibile dal fatto che il registro viene suddiviso negli 8 bit che lo compongono, ammettendo poi la possibilità di usare separatamente ciascuno di questi bit. In seguito incontrerete molte volte la frase "il segnale è stato alzato" oppure "abbassato". Si tratta di un riferimento ad un singolo bit che può venire "alzato", cioè posto uguale a 1, oppure "abbassato", cioè posto uguale a 0. Questo bit viene dunque chiamato "segnale di riporto" (ovvero, in inglese, "carry flag"). La caratteristica che rende particolarmente utile il segnale di riproto è che, ogni volta che due registri vengono sommati assieme e il risultato è superiore a 255, allora esso viene "alzato" (posto uguale a 1). Se invece il risultato non supera il limite di 255, il segnale di riporto viene "abbassato" (posto uguale a 0). Non possiamo accedere direttamente al registro F, e quindi non è possibile cambiare direttamente lo stato dei segnali; è però abbastanza semplice accedere al contenuto di questi bit.

È possibile sommare al registro A qualsiasi altro registro, incluso se stesso, ma non si può sommare un registro singolo ad un altro registro diverso da A. Nel corso del vostro studio del linguaggio macchina troverete che il registro A, che spesso viene chiamato "accumulatore", è di utilizzo molto frequente. Non c'è un motivo particolare per questo fatto, tranne che questa è la prima lettera dell'alfabeto e anche l'iniziale della parola accumulatore, e quindi è molto facile ricordarsene. Molti comandi importanti usano questo registro, così come il registro HL è il più usato nel caso di coppie di registri. Al registro A è possibile sommare il contenuto di un altro registro oppure un numero, ricordando però che questo numero deve essere compreso fra 0 e 255. Di seguito viene riportata una lista di tutti i comandi di somma e i loro corrispettivi codici in linguaggio macchina. Questi codici sono in successione, quindi dovrebbe essere abbastanza semplice ricordarli.

OP CODE	HEX	DECIMAL
ADD A,(HL)	86	134
ADD A,A	87	135
ADD A,B	80	128
ADD A,C	81	129
ADD A,D	82	130
ADD A,E	83	131
ADD A,H	84	132
ADD A,L	85	133
ADD A,xx	C6xx	198,XX

La somma di registri singoli è molto simile alla somma di coppie di registri, come mostrato dagli esempi che seguono:

Esempio 1:

```
org 30000
30000 3E 05          ld a,5
30002 06 06          ld b,6
30004 80             add a,b
30005 4F             ld c,a
30006 06 00          ld b,0
30008 C9             ret
```

Esempio 2:

```
org 30000
30000 3E 0E          ld a,14
30002 C6 06          add a,06
30004 4F             ld c,a
30005 06 00          ld b,0
30007 C9             ret
```

Nell'esempio 1, le prime due istruzioni servono semplicemente per caricare i numeri 5 e 6 nei due registri A e B rispettivamente. La terza istruzione somma il contenuto del registro B a quello del registro A e il risultato rimane in A. Le due righe successive ci consentono poi di far stampare la risposta usando il semplice comando "PRINT USR" del BASIC.

Nel secondo esempio stiamo sommando 6 al contenuto dell'accumulatore, ovvero nel registro A, per poi caricare in C il contenuto di A e in B il numero 0; infine c'è il comando "RET", cosicché il numero mostrato sul video dopo il comando "PRINT USR" sarà il contenuto di BC, cioè il risultato della somma A+6.
Un'istruzione "ADD" influenza sempre il segnale di riporto: come si è detto precedentemente, se non c'è riporto il segnale viene posto uguale a zero, se c'è riporto esso viene posto uguale ad uno.

Finora non abbiamo potuto fare uso in alcun modo di questo segnale. Il modo più semplice per farlo è tramite il comando "ADC", che significa "somma con riporto" (in inglese "Add with Carry"). Quando la macchina arriva all'istruzione "ADC A,B" prende il contenuto del registro B, lo somma a quello del registro A e lascia il risultato memorizzato nel registro A, come con il precedente comando "ADD A,B". Questo comando, però, aggiunge a questo risultato anche il valore del segnale di riporto: poiché il segnale non è ancora stato modificato per questa istruzione, il numero che viene sommato al risultato è il valore del segnale prima di questa istruzione. Dopo aver fatto tutto ciò la macchina memorizza il risultato

nel registro A, influenzando il segnale di riporto. Quindi, se il segnale era stato fissato da una prepcedente addizione nel programma e da allora non è stato cambiato, il risultato di un comando "ADC" sarà influenzato dallo stato del segnale. A prima vista ciò può sembrare più una seccatura che un aiuto nella programmazione. Tuttavia, se ripensate ai vostri tempi di scuola elementare, quando avete imparato a fare le addizioni, vi diverrà subito evidente che questa è una possibilità molto utile: essa non è altro che il principio del "riporto" nelle addizioni. Sappiamo che per prima cosa si sommano i numeri della colonna di destra, e se c'è un "overflow" allora un uno deve essere sommato alla colonna adiacente, a sinistra. Per esempio, se sommiamo i numeri 14 e 7 dobbiamo prima sommare 4 e 7, ottenendo il risultato 11: scriviamo 1 sotto la colonna di destra e "riportiamo 1". Sommiamo quindi le cifre della colonna di sinistra, cioè 1 più 0 e vi sommiamo anche il numero che è stato riportato, in questo caso 1: otteniamo così il risultato 2, che scriviamo sotto la colonna di sinistra. Adesso possiamo leggere il risultato da sinistra a destra ed avere 21.

L'esempio che segue mostra come possiamo usare questa funzione in un programma in linguaggio macchina. Poiché il programma è piuttosto lungo e complicato, non preoccupatevi della spiegazione fino a quando non l'avrete caricato e provato: poi troverete la spiegazione nel paragrafo successivo. Per ora assicuratevi soltanto di inserire il programma esattamente come lo vedete scritto.

```
org 30000
30000 16 33       ld d,51
30002 1E 85       ld e,133
30004 26 7B       ld h,123
30006 2E C7       ld l,199
30008 7D          ld a,l
30009 83          add a,e
30010 6F          ld l,a
30011 7C          ld a,h
30012 8A          adc a,d
30013 67          ld h,a
30014 44          ld b,h
30015 4D          ld c,l
30016 C9          ret
```

Il comando 1 assegna il valore 51 al registro D, il comando 2 assegna il valore 133 al registro E; il comando 3 assegna il valore 123 al registro H, e il comando 4 assegna il valore 199 al registro L.
L'obiettivo di questo programma è sommare il contenuto della coppia di registri DE a quello della coppia HL. Questa operazione viene eseguita non con un solo comando, ma sommando le due metà di ogni coppia. In altre parole, ciò che vogliamo fare è prima sommare E con L, e poi sommare D con H. Il primo problema che incontriamo è il fatto che possiamo sommare un registro soltanto

con il registro A, quindi è necessario sostituire il registro L con il registro A: ciò viene fatto facilmente con il comando S, che carica il contenuto di L nel registro A. Dopodiché possiamo sommare E con A. Adesso il valore di A è uguale al contenuto di L più il contenuto di E; noi, però, vogliamo ottenere questa risposta non nel registro A, ma nel registro L; superiamo semplicemente questo problema con il caricamento inverso del contenuto di A in L.

Poiché abbiamo già sommato i due byte meno significativo con il comando 6, il segnale di riporto dovrebbe essere già stato posto ad uno se c'era "overflow": infatti è proprio così. Adesso ripercorriamo quasi la stessa procedura per sommare i due byte più significativo, cioè i byte D ed H.
Questa volta, però, usiamo il comando "ADC", somma con riporto. In questo modo sommiamo D con A, come abbiamo fatto prima; questa volta, però, abbiamo sommato anche il valore del segnale di riporto. In questo modo, se quando abbiamo sommato L ed E il risultato era maggiore di 255, adesso aggiungeremo un 1 extra al valore dei byte più significativo. Il risultato che otteniamo con questa procedura è dunque esatto. I comandi 11 e 12 sono semplicemente i nostri soliti comandi, che ci consentono di caricare il contenuto di HL in BC, in modo da ottenere la stampa sul video dopo l'istruzione "PRINT USR" del BASIC. Un'altra cosa da notare in questo programma è che i due comandi che seguono "ADD A,E" e precedono "ADC A,D" non influenzano il contenuto del segnale di riporto; sappiamo così cosa aspettarci. Per questo motivo alla fine del libro trovate una lista di tutti i comandi con l'indicazione degli effetti sul segnale di riporto.

Ormai dovreste essere in grado di capire la differenza fra "ADD" e "ADC". Eccovi ora la lista dei codici e delle mnemoniche per le varie combinazioni.

OP CODE	HEX	DECIAML
ADC A,(HL)	8E	142
ADC A,A	8F	143
ADC A,B	88	136
ADC A,C	89	137
ADC A,D	8A	138
ADC A,E	8B	139
ADC A,H	8C	140
ADC A,L	8D	141
ADC A,xx	CExx	206,XX
ADC HL,BC	ED4A	237,74
ADC HL,DE	ED5A	237,90
ADC HL,HL	ED6A	237,106

Non è possibile sommare direttamente una costante alla coppia di registri HL; però ciò può essere facilmente ottenuto caricando in una coppia di registri il numero che si vuol sommare ad HL e poi sommando quella coppia ad HL. Il risultato in HL sarà pari al valore di HL più la costante. Per esempio:

```
org 30000
30000 11 39 00      ld de,57
30003 19            add hl,de
```

Questo metodo ha lo svantaggio di richiedere l'uso della coppia DE, che può essere necessaria per altri motivi. Un altro modo di ottenere questo stesso obiettivo viene mostrato nel prossimo esempio, nel quale l'unico registro usato, oltre ad HL, è A. Notate che in questo esempio ho usato di nuovo il comando di somma con riporto proprio come abbiamo fato nel primo esempio: in questo modo se il byte meno significativo va in "overflow" ci sarà riporto nell'altro byte, ed il risultato sarà esatto.

```
org 30000
30000 7D            ld a,l
30001 C6 39         add a,57
30003 6F            ld l,a
30004 7C            ld a,h
30005 CE 00         adc a,0
30007 67            ld h,a
```

Immagino che per ora abbiate abbastanza lavoro di apprendimento del linguaggio macchina; eccovi quindi un programma da caricare e usare. Sebbene esso contenga alcuni dei comandi che abbiamo già studiato, non è essenziale che siate in grado di comprenderlo, in questa fase, quindi non preoccupatevi e non arrabbiatevi se c'è ancora qualcosa di misterioso. Quando sarete arrivati alla fine del libro potrà esservi utile tornare indietro e scoprire come veramente funziona. L'idea di inserire a questo punto questo programma è quella di permettervi di usarlo, di vedere gli effetti di queste istruzioni in linguaggio macchina e di fare un breve intervallo.

```
org 30000
30000 21 00 40      ld hl,16384
30003 06 C0         ld b,192
'CAMBIARE B & HL PER ALTRI
'BLOCCHI DI SCHERMO
A
30005 0E 20         ld c,32
B
30007 5E            ld e,(hl)
30008 CB 1E         rr (hl)
30010 23            inc hl
```

```
30011 0D                dec c
30012 20 F9             jr nz,B
30014 CB 43             bit 0,e
30016 28 09             jr z,C
30018 E5                push hl
30019 11 1F 00          ld de,31
30022 ED 52             sbc hl,de
30024 CB FE             set 7,(hl)
'RES 7,(HL) PER NON RITORNO'
30026 E1                pop hl
C
30027 A7                and a
30028 10 E7             djnz A
30030 C9                ret
```

Come preludio alla sezione successiva del libro, guardate attentamente questo listato e notate tutti i comandi che hanno parentesi tonde.

Il caricamento del contenuto di una locazione di memoria in un registro

Prima, come forse ricorderete, abbiamo visto come i numeri possano essere immagazzinati in "scatole" o locazioni di memoria. Vi ricorderete anche che ciascuna di queste locazioni ha un numero che costituiva l'indirizzo specifico della scatola. Il motivo per cui ci sono questi indirizzi è che così possiamo accedere al contenuto della scatola con relativa facilità. Ho già accennato quanto sarebbe utile poter manipolare i numeri delle scatole, ma finora non abbiamo parlato di come si possano effettivamente estrarre i numeri dalle scatole, ovvero di come prendere un numero da una locazione di memoria e caricarlo in un registro. Ovviamente quando esso si troverà nel registro saremo in grado di manipolarlo come vogliamo. Dovrebbe inoltre essere possibile caricare il contenuto dell'accumulatore, o di qualsiasi altro registro in una locazione di memoria. In questo modo diventa possibile conservare un particolare numero per un successivo utilizzo. Inoltre, usando questo metodo è possibile avere una quantità molto grande di numeri, tutti ad accesso immediato.

Consideriamo ciò che vogliamo effettivamente fare. Il nostro compito è caricare, per esempio nell'accumulatore, cioè il registro A, il contenuto di una certa locazione. Questa locazione ha un indirizzo. Se scrivessimo semplicemente "load A" seguito dall'indirizzo, il computer scambierebbe l'indirizzo per una costante numerica. Per esempio, se scrivessimo "LD A,12" sarebbe ragionevole attendersi che il computer pensi di caricare nell'accumulatore il numero 12. Ciò che vogliamo invece è caricare nell'accumulatore il contenuto di una particolare locazione di memoria. In effetti questo si può ottenere in modo molto semplice: invece di scrivere il numero dopo il comando, scriviamo l'indirizzo della locazione desiderata

fra parentesi dopo la virgola. Per esempio "LD A,(2465)". Il computer adesso carica nell'accumulatore il valore contenuto nella locazione 2465. Poiché ciascuna locazione contiene solo numeri fra 0 e 255, non sorge alcun problema nel caricare il contenuto di una locazione in un registro ad un solo byte. Se, per esempio, nella locazione 2465 fosse contenuto il numero 64, dopo l'esecuzione di questo comando l'accumulatore conterrebbe anch'esso il valore 64. Come per tutti questi comandi il contenuto della locazione rimarrebbe inalterato, cioè nella locazione ci sarebbe ancora 64. Questo è un comando molto utile, che ci consente di prendere il contenuto di una qualsiasi locazione di memoria, metterlo in un registro e manipolarlo a nostro piacimento. Per esempio:

```
org 30000
30000 3A 30 75      ld a,(30000)
30003 C6 29         add a,41
```

L'altro problema adesso è di come rimettere il numero manipolato nel vecchio indirizzo oppure in un altro indirizzo. Ciò può essere effettuato usando una variante del comando precedente. Pensando a cosa effettivamente vogliamo fare a questo punto, è chiaro che, invece di caricare il contenuto del nostro indirizzo nell'assemblatore, vogliamo caricare il contenuto dell'accumulatore nell'indirizzo. Quindi useremo a rovescio il comando del caricamento: "LD (indirizzo), A". Ecco subito un esempio:

```
org 30000
30000 C6 0F         add a,15
30002 32 30 75      ld (30000),a
30005 C9            ret
```

Riconsiderando tutta questa procedura,cioé manipolarlo e quindi rimetterlo nella medesima locazione prendere un numero da una locazione, possiamo vedere ora un programma dimostrativo che cambia il colore del video: esso funziona prendendo il codice del colore che già c'è e modificandolo in modo da ottenere un diverso colore. Poiché il video, per quanto riguarda il colore, richiede 704 byte, è necessario ripetere questa operazione 704 volte. Per fortuna è possibile usare un'altra istruzione, cosicché non dobbiamo scrivere il programma 704 volte! Vedremo dettagliatamente questa istruzione più avanti.
Ora, prima di procedere con l'esempio del programma di cambiamento del colore, ecco un piccolo compito per voi: scrivete una routine che prende la codifica di un qualsiasi carattere dalla memoria, la cambia con quella dell'inverso di quel carattere (cioè nero su bianco invece che bianco su nero) e poi lo rimette al suo posto. Un piccolo suggerimento: il codice per l'inverso di ogni carattere è dato

dal codice del carattere normale più 128 (decimale). Basta quindi sommare 128 al codice di ogni carattere per ottenere il relativo carattere inverso.

```
org 30000
30000 21 00 58    ld hl,22528
30003 01 C0 02    ld bc,704
30006 1E 06       ld e,6
LOOP
30008 7E          ld a,(hl)
30009 CB 87       res 0,a
30011 CB 8F       res 1,a
30013 CB 97       res 2,a
30015 83          add a,e
30016 77          ld (hl),a
30017 23          inc hl
30018 0B          dec bc
30019 79          ld a,c
30020 FE 00       cp 0
30022 20 F0       jr nz,LOOP
30024 78          ld a,b
30025 FE 00       cp 0
30027 20 EB       jr nz,LOOP
30029 C9          ret
```

NOTA: ld e, "nuovo colore INK"

Sottrazione

In linguaggio macchina le istruzioni per la sottrazione di numeri e di registri da un altro registro sono esattamente uguali alle corrispondenti istruzioni per l'addizione. Per questo motivo la sottrazione è molto semplice da capire. Chiaramente però ci sono alcune differenze. Con l'addizione si può andare in overflow, con la sottrazione si va in "underflow", cioè il numero minuendo è più piccolo del sottraendo. Se si sottrae 11 da 5 ci sarà un underflow di 6. Se nella sottrazione si verifica un underflow, il segnale di riporto viene posto uguale a 1, altrimenti esso viene azzerato. L'altra cosa da ricordare è che non c'è un segnale per la positività o negatività di un numero. Quando nell'addizione si verifica un overflow i numeri ricominciano a sommarsi da zero: per esempio eseguendo 5 più 255, invece di andare a 256 e poi sommare ancora 4, il computer tornerà a zero e poi sommerà 4. Nella sottrazione invece esso tornerà a 256 e poi farà la sottrazione.

Il comando per la sottrazione funziona come quello per l'addizione, nel senso che il comando "SUB A,B" (SUB è l'abbreviazione di "subtract") sottrae il valore del registro B da quello del registro A, e il risultato viene memorizzato nel registro A. Il segnale di riporto viene opportunamente "alzato" (posto uguale a 1) o "abbassato" (posto uguale a 0).

Poiché il comando SUB si riferisce a registri ad un solo byte, e soltanto con l'accumulatore come minuendo, è abbastanza comune scrivere "SUB B" invece di "SUB A,B": ciò può essere causa di confusione all'inizio, ma vi abituerete presto a questa scrittura. Dato che la sottrazione non è un argomento molto semplice quando lo si tratta in un formato poco familiare, può essere utile a questo punto rivedere il lavoro fatto sulla sottrazione all'inizio di questo libro, in modo da fissare meglio le idee. I vari comandi assembler ed i loro equivalenti in linguaggio macchina sono indicati di seguito:

OP CODE	HEX	DECIMAL
SUB (HL)	96	150
SUB A	97	151
SUB B	90	144
SUB C	91	145
SUB D	92	146
SUB E	93	147
SUB H	94	148
SUB L	95	149
SUB xx	D6xx	214,XX

Ecco un breve esempio su come usare il comando di sottrazione:

```
org 30000
30000 3E 0B        ld a,11
30002 06 07        ld b,7
30004 97           sub a,b
30005 4F           ld c,a
30006 06 00        ld b,0
30008 C9           ret
```

È possibile sottrarre costanti numeriche dal registro A. Per esempio, l'istruzione "SUB A,100" serve a sottrarre 100 dal contenuto di A. Il risultato viene poi conservato in A. Notate che, mentre esistono istruzioni di somma di coppie di registri, non esistono invece i comandi di sottrazione fra coppie. Ecco ora un altro esempio di sottrazione:

```
ORG 30000
30000   0600    LD B,00
30002   3E58    LD A,88
30004   D633    SUB 51
```

```
30006  4F     LD C,A
30007  C9     RET
```

La sottrazione con il riporto invece ("Subtract with Carry", SBC) funziona anche fra coppie di registri, però anche qui, come con ADD e ADC, si possono usare solo la coppia HL e il registro singolo A. Il comando "SBC A,C" sottrae il valore del registro C da quello del registro A, e poi sottrae il valore del segnale di riporto da questo risultato memorizzandolo infine nel registro A. Il comando di "sottrazione con riporto" può essere usato quando si deve fare una sottrazione fra due numeri per i quali si può incorrere nell'underflow.

Ripensando al modo in cui abbiamo imparato per la prima volta a fare le sottrazioni, possiamo capire come ciò funzioni. Supponiamo di voler eseguire 21 meno 19: ecco come si procede. Prima si sottrae 9 da 1: ciò non è propriamente possibile mantenendo il risultato positivo, quindi si deve "prendere a prestito" 1 dalla colonna a sinistra (cioè in questo caso si prende a prestito una decina). Adesso possiamo sottrarre 9 da 11, e otteniamo 2 come risultato. Passando alla colonna successiva si deve ricordare che essa è stata ridotta di 1 per effetto di ciò che è stato chiamato underflow nella colonna precedente. Bisogna allora sottrarre 1 da 2, con risultato 1; però dobbiamo sottrarre anche il segnale di riporto dal risultato, ottenendo 0, e la risposta finale è 2.

Dalla lista che segue potete notare il comando SBC applicato ad una coppia di registri richiede due byte, oppure un solo byte se applicato ad un registro singolo: questi comandi sono dunque veramente molto sintetici. Ecco ora la lista dei codici assembler e in linguaggio macchina:

OP CODE	HEX	DECIMAL
SBC A,(HL)	9E	158
SBC A,A	9F	159
SBC A,B	98	152
SBC A,C	99	153
SBC A,D	9A	154
SBC A,E	9B	155
SBC A,H	9C	156
SBC A,L	9D	157
SBC A,xx	DExx	222,XX
SBC HL,BC	ED42	237,66
SBC HL,DE	ED52	237,82
SBC HL,HL	ED62	237,98

Operazioni con il segnale di riporto

Spesso, quando si usano le istruzioni di "sottrazione con riporto" e di "addizione con riporto", è molto utile essere in grado di determinare il contenuto del segnale di riporto. Con questi comandi infatti il risultato può essere alterato, ed il programma fatto "impazzire", semplicemente perché il segnale era fissato a 1 quando invece sarebbe dovuto essere a 0.
Sfortunatamente non è possibile azzerare direttamente questo segnale, ma ciò può essere ottenuto con un semplice trucco. Sappiamo che ogni volta che si somma un numero all'accumulatore di riporto viene fissato a seconda che ci sia o meno overflow. Se allora sommiamo 0 all'accumulatore non è possibile che si verifichi overflow, e quindi facendo proprio questa operazione possiamo azzerare il segnale di riporto. Ciò succede proprio perché il segnale viene posto uguale a zero se non si ha overflow. Probabilmente questo è il più utile comando con il segnale di riporto. Porre il segnale uguale a 1 è invece molto più semplice, dato che esiste un'istruzione proprio per questo scopo. L'istruzione è SCF ("Set Carry Flag"), e il suo corrispettivo in linguaggio macchina è 37 esadecimale.

Si conclude così il capitolo. Abbiamo visto due modi molto importanti di manipolare i registri, e i modi in cui possiamo prendere i numeri contenuti nelle locazioni di memoria per metterli nei registri. Abbiamo anche visto alcuni comandi più complicati e il modo di superare i problemi circa la fissazione del valore del segnale di parità. Spero che a questo punto siate in grado di scrivere per conto vostro alcuni brevi programmi in linguaggio macchina che facciano uso di questi comandi.

CAPITOLO 6

ASSEMBLATORI, DISASSEMBLATORI E CORREZIONE DEI PROGRAMMI

In questo capitolo faremo una sosta nell'apprendimento del linguaggio macchina vero e proprio. Vedremo invece i tre tipi principali di programmi che sono stati scritti allo scopo specifico di aiutare il programmatore in linguaggio macchina. Ora che avete già imparato diverse cose su come scrivere semplici programmi in linguaggio macchina, inizierete presto a cimentarvi con programmi più sofisticati. Ciò richiederà scrivere routine in linguaggio macchina più lunghe di quelle viste finora, e sarà pertanto utile avere alcuni strumenti che facilitino il lavoro. La maggior parte di questi strumenti sono scritti anch'essi in linguaggio macchina. È chiaro che il programmatore che li ha scritti ha una notevole conoscenza in questa materia, ma non è detto che sia in grado di spiegare in modo semplice le istruzioni per il funzionamentò di tali programmi. Perciò il principiante può trovarsi in serie difficoltà quando deve decifrare i manuali per i programmi di correzione dell'assembler ("debugger") o per i disassemblatori.
Alla fine di questo capitolo, proprio per facilitare questo libro, ho inserito una breve sezione in cui si spiega esattamente come usare un esempio di ciascun tipo di questi programmi. Si noti che questi esempi sono tutti disponibili sul mercato: molti li potrete ottenere nei negozi della vostra città, ma comunque alla fine del libro potrete trovare una lista di inidirizzi da cui poter ordinare per posta questi programmi.
La scelta dei programmi di servizio è spesso una questione di preferenze e di esperienze personali. I programmi usati negli esempi di questo libro sono stati tutti assemblati o disassemblati con l'aiuto di programmi della ACS Software.

Assemblatori

A questo punto è importante che capiamo esattamente cosa si intende per linguaggio macchina e linguaggio assembly, e che comprendiamo bene tutto ciò che è stato detto finora. Ecco quindi un breve riepilogo (se vi sentite sicuri di questi concetti potete benissimo saltare questa sezione).

Il "linguagio macchina" è costituito unicamente da numeri. Il computer comprende questi numeri e li esegue come comandi. Tuttavia, dover ricordare una gran quantità di numeri ed i rispettivi comandi cui ciascuno di essi si riferisce non è particolarmente facile per noi esseri umani. Per questo motivo è stato creato il "linguaggio assembly": ciascun comando di questo linguaggio corrisponde univocamente ad un comando in linguaggio macchina; in assembly, inoltre, le istruzioni sono scritte con mnemoniche che sono le abbreviazioni dei comandi effettivi. Per esempio, se vogliamo eseguire il comando "carica nel registro A il contenuto del registro B", ciò può essere ottenuto in linguaggio macchina con la semplice immissione del codice 78, e con la sua esecuzione. Non è comunque facile ricordare il numero 78, e la sua corrispondenza con questa istruzione. Un modo di ovviare a questo problema è quello di tenersi a portata di mano una lista con tutte le istruzioni ed i relativi codici, in modo da poter trovare il comando desiderato ed il corrispondente codice in linguaggio macchina.

Il primo problema che si ha con questo metodo di lavoro è che se la lista contenesse una spiegazione dettagliata di tutti i comandi, risulterebbe di notevoli dimensioni. Al posto di questi vengono invece usate le cosiddette "mnemoniche": esse non sono altro che l'abbreviazione delle parole inglesi relative ai vari comandi.

Invece di dire "carica (load) nel registro A il contenuto del registro B" diremo semplicemente "LD A,B". Vi sarete già resi conto che questa abbreviazione si ricorda più facilmente che non il codice del linguaggio macchina; essa, inoltre, richiede molto meno spazio della spiegazione completa del comando. Ora, se volessimo usare il metodo della tabella di riferimento, potremmo farci semplicemente la lista di tutte le menemoniche, con a fianco il relativo codice in linguaggio macchina (questo è quanto è stato fatto alla fine di questo libro). Se poi imparassimo tutte le mnemoniche per le varie istruzioni del linguaggio macchina, avremmo il cosiddetto "linguaggio assembly". In altre parole, il linguaggio assembly è formato proprio dalle mnemoniche del linguaggio macchina. Per questo motivo si può vedere che il linguaggio assembly non è un adattamento del linguaggio macchina, come lo è ad esempio il BASIC, e che c'è un'unica mnemonica per ogni particolare istruzione in linguaggio macchina, e viceversa.

Ciò che abbiamo ottenuto a questo punto è un linguaggio che possiamo facilmente comprendere ed usare. Tuttavia, anche se ora abbiamo un linguaggio più semplice da usare, esso richiede ancora la spesa di tempo e fatica per la "conversione". Perché allora non rivolgersi al buon vecchio computer per fargli svolgere tutto il faticoso lavoro di conversione dalle mnemoniche ai codici del linguaggio macchina? Questo è quanto l'assembler riesce a fare per noi: è un programma che "assembla", cioè traduce le mnemoniche in codici di linguaggio macchina.

Un altro modo di vedere questo passaggio è quello di considerare come funziona

un ristorante "fast food". In uno di questi locali fate le vostre ordinazioni sulla base di un menù, sul quale sono elencati i vari piatti, per esempio hamburger e così via. Quando poi date l'ordinazione al cameriere, questo si annoterà non l'effettiva ordinazione, ma il numero che corrisponde a quella determinata pietanza. Ciò che egli sta facendo, quindi, non è altro che "assemblare" la vostra ordinazione in forma tale che le persone in cucina possano capirla ed usarla più rapidamente.

Tutti i comuni assembler per lo Spectrum lavorano sulla base del principio che prima si scrive il programma in linguaggio assembler e poi, quando lo si è completato, si dà ordine all'assembler di convertirlo in linguaggio macchina. Il codice così prodotto viene riposto in una certa zona di memoria preventivamente specificata a proprio piacimento. Naturalmente si deve anche tener conto della quantità di memoria ancora disponibile nel computer. Un programma assembler è un tipo di programma piuttosto complesso, ed alcuni degli assemblatori esistenti sul mercato sono adatti solo allo Spectrum 48K. Ciò ha influenzato la mia scelta di un assemblatore per la stesura di questo libro, dato che i programmi della ACS funzionano sia sulla versione di 16K che su quella a 48K.

Sfogliando una qualsiasi rivista del settore troverete numerosi annunci pubblicitari di assemblatori. Il loro prezzo è leggermente più elevato di quello di un programma di gioco. Ciò è dovuto a due motivi: anzitutto essi sono per la maggior parte più complicati dei giochi; in secondo luogo essi non sono prodotti di massa, e non si può quindi sperare di venderne in quantità molto elevate. Comunque, credetemi, vale veramente la pena di acquistarli: essi sono di incredibile aiuto per il serio programmatore in linguaggio macchina.

Disassemblatori

Un disassemblatore compie esattamente l'operazione inversa rispetto ad un assemblatore: converte infatti il codice in linguaggio macchina nelle mnemoniche del linguaggio assembly.

Due sono i modi di usare un disassemblatore:

1. Semplicemente come uno strumento per controllare che quel che si è scritto nella memoria del computer corrisponda esattamente a ciò che si voleva scrivere.

2. Per vedere che cosa altre persone hanno scritto in linguaggio macchina, e capire come hanno risolto certi problemi.

Per esempio potreste disassemblare alcune parti della ROM dello Spectrum per capire meglio cosa succede lì dentro e, se possibile, usare alcune delle sue routine. In molte riviste, inoltre, quando viene dato il listato di un programma in

linguagio macchina esso appare solo con i codici esadecimali. Vi consiglio di caricare programmi di altre persone e quindi disassemblarli, in modo da farvi delle idee su come altri scrivono i programmi in linguaggio macchina. In questo modo si può imparare davvero molto.

Con i programmi assemblatore e disassemblatore usati per questo libro è possibile, se si ha a disposizione una macchina a 48K, il caricamento contemporaneo. Di nuovo, quando scegliete i programmi da acquistare, tenete presente l'utilità di questo tipo di compatibilità.
In una macchina a 16K non è possibile caricare contemporaneamente assemblatore e disassemblatore, semplicemente perché non c'è spazio a sufficienza per entrambi. In questi casi è possibile caricarne solo uno alla volta. Poiché comunque si possono caricare e memorizzare i programmi in linguaggio macchina indipendentemente dai programmi già esistenti nel computer, potremo assemblarli usando il programma assembler, conservarli per esempio su nastro e poi ricaricarli insieme al disassembler per il disassemblaggio.

I programmi di correzione ("debugger")

Il terzo ed ultimo tipo di programmi che consideriamo non può essere usato quando si sta iniziando a scrivere un programma in linguaggio macchina; viene invece usato per trovare gli errori e modificare il programma dopo che questo sia stato assemblato. Quando si compra un programma di "debug" non si compra in genere un solo programma. Un programma di debug per il linguaggio macchina che sia abbastanza buono offrirà senz'altro alcune delle seguenti caratteristiche:

1. L'opzione che consente di fare eseguire la vostra routine in linguaggio macchina un'istruzione alla volta e di mostrare il contenuto di tutti i registri dopo ogni passo. Questa possibilità è estremamente utile quando si debba ricercare un errore in un programma, oppure quando si voglia vedere come funziona esattamente una routine scritta da un'altra persona.

2. La possibilità di inserire un "punto d'arresto" nel vostro programma, cioè di far girare il programma fino a quel punto e poi fermarlo. Questo comando può essere usato in vari modi. Può darsi che non vi interessi conoscere il contenuto dei registri dopo ogni passo del programma, specialmente se esso è piuttosto lungo, ma che vogliate conoscere lo stato dei registri e dei segnali solo dopo che sia stata completata una certa operazione o blocco di operazioni. Dovrebbe poi essere abbastanza semplice far riprendere l'esecuzione del programma, di solito con la semplice pressione di un determinato tasto.

3. Essere in grado di modificare il contenuto di un particolare registro mentre il programma è fermo ad un "punto di arresto", cosa che di solito può essere fatta molto facilmente. Questo può essere utile in due casi particolari.
Anzitutto, se desiderate sapere quale sarebbe il risultato di una certa operazione nel caso in cui nei registri vengano caricati particolari numeri. In secondo luogo può essere utile avere la possibilità di simulare una certa condizione, per esempio quella in cui i numeri inseriti nel computer sono troppo grandi per essere usati dal programma, in modo da poter verificare il funzionamento delle proprie routine di gestione degli errori.

4. La capacità di eseguire un programma senza dover prima tornare al BASIC. Non sempre è facile, infatti, tornare al BASIC ed usare il comando USR.

5. La possibilità di convertire i numeri da esadecimali a decimali nell'ambito del programma di debug.

6. Tre utili comandi che usati insieme consentono al programma di debug di modificare il vostro programma in linguaggio macchina già assemblato. Essi permettono l'ispezione del contenuto di un qualsiasi indirizzo e la sua eventuale modifica, se necessario. Inoltre con molti prograsmmi di debug, se si tiene premuto il tasto ENTER, il contenuto degli indirizzi successivi viene automaticamente visualizzato sul video, senza inserire nuovi valori, a meno che, ovviamente, non scriviate un numero prima di premere ENTER. Se poi vi accorgete di dover inserire un comando all'interno di un programma in linguaggio macchina già assemblato, la cosa risulta abbastanza complicata, dato che si tratterebbe di muovere verso l'alto o verso il basso una metà del programma. Con molti programmi "debugger", invece, l'inserimento di una nuova istruzione è semplice: basta dire al computer esattamente dove si vuole effettuare l'inserimento, e quanti byte si vogliono inserire, in modo che lo spazio necessario possa venire creato; poi si potranno inserire i codici voluti. L'istruzione complementare a questa è naturalmente quella che permette di cancellare un certo comando: anch'essa può essere trovata in molti programmi debugger, e funziona esattamente al contrario del programma di inserimento.

Questa è solo una breve guida ad alcune delle cose che potrete trovare in un programma di correzione degli errori, anche se di solito troverete molti altri comandi oltre a quelli elencati. Il modo in cui questi programmi funzionano cambia a seconda del particolare programma che state usando.

Spesso un assembler o un disassembler possono essere inclusi in un debugger, o viceversa. Per esempio, spesso troverete che un debugger ha anche una funzione di disassemblaggio. Un altro esempio di questa pratica è l'assembler fornito dalla Picturesque. Oltre a permettere operazioni di disassemblaggio, esso

consente anche di modificare i programmi, usando vari comandi sofisticati. Il programma di monitor, che è essenzialmente un programma debugger con assembler, può essere usato contemporaneamente al programma assemblatore/editor, ottenendo così una combinazione molto potente.
Allo scopo di semplificare le cose, e anche per consentire qualche risparmio, i programmi assembler, disassembler e debugger usati per questo libro sono tutti programmi separati, disponibili su cassette distinte.

Un semplice programma di debug non deve necessariamente essere scritto in linguaggio macchina. Può essere che decidiate di non voler ancora ricorrere all'acquisto di un programma. Il miglior compromesso è allora quello di usare il programma di monitor listato nelle pagine seguenti. Esso vi permetterà di inserire nel computer il programma in linguaggio macchina e di compiere soddisfacenti operazioni di editor, anche se dovrete inserirlo con i codici esadecimali, non in assembler. La conversione delle mnemoniche del linguaggio assembler potrà invece essere fatta usando le appendici alla fine del libro.

Quando avrete scritto questo semplice programma ricordatevi di conservarlo su cassetta prima di provarlo, e non appena sarete sicuri che funziona correttamente fatene una copia separata di riserva, su di un'altra cassetta, nell'eventualità che una delle due cassette risulti in qualche modo danneggiata.

```
   5 CLEAR 31999
  10 PRINT "    ****Spectrum Moni
tor****"'''"        by James Walsh
'83"
  30 REM Load M C
 100 INPUT "Indirizzo di inizio
(dec)?";loc
 110 PRINT AT 5,0;
 120 PRINT loc;"...";
 130 LET n=PEEK loc: GO SUB 1400
 140 INPUT z$
 145 PRINT "...";z$
 150 IF z$="fine" OR z$="FINE" T
HEN GO TO 800
 160 IF z$="p" OR z$="P" THEN CO
PY : GO TO 120
 170 IF z$="S" OR z$="s" THEN GO
 TO 280
 190 IF z$="v" OR z$="V" THEN GO
 SUB 1600: GO TO 120
 200 IF z$="j" OR z$="J" THEN LE
T loc=loc-1: GO TO 120
 210 IF z$="n" OR z$="N" THEN GO
 TO 100
 220 IF z$="z" OR z$="Z" THEN LE
T z$="00"
 230 LET ans=(CODE z$(1)-48)*16
```

```
 240 IF ans>9*16 THEN LET ans=an
s-7*19
 250 LET l=CODE z$(2)-48: IF l>9
 THEN LET l=l-7
 260 LET ans=ans+l
 270 POKE loc,ans
 280 LET loc=loc+1
 290 GO TO 120
 800 STOP
1400 REM **HEX/DEC subroutine**
1410 LET x=INT (n/16)
1420 LET y=((n/16)-INT (n/16))*1
6
1430 IF x>9 THEN LET x=x+7
1440 IF y>9 THEN LET y=y+7
1450 LET x=x+48: LET y=y+48
1460 PRINT CHR$ x;CHR$ y;
1470 RETURN
1600 REM **INPUT HEY/DEC SUB.**
1605 INPUT "Numero da convertire
";n
1607 LET a=n
1610 LET x=INT (n/16)
1620 LET y=((n/16)-INT (n/16))*1
6
1630 IF x>9 THEN LET x=x+7
1640 IF y>9 THEN LET y=y+7
1650 LET x=x+48: LET y=y+48
1660 PRINT a;" = ";CHR$ x;CHR$ y
1670 RETURN
```

Vediamo ora come si usa questo programma di monitor. Esso è più breve dei programmi commerciali che abbiamo visto finora, ed è certamente il più semplice da usare. La prima cosa da notare è che la riga 5 è usata per specificare il punto di RAMTOP per mezzo dell'istruzione CLEAR. È quindi necessario modificare questo valore a seconda della parte di memoria che si vuole usare per la routine in linguaggio macchina. Adesso fate girare il programma. Vi sarà chiesto di inserire il valore della locazione di inizio: fatelo. Questo indirizzo iniziale comparirà sullo schermo seguito dal suo contenuto in forma esadecimale. A questo punto si potrà inserire un nuovo valore (in esadecimale) ricordando di usare solo lettere maiuscole. In alternativa può essere usato uno dei seguenti comandi:

P - Copia il contenuto dello schermo sulla stampante.
S - Non modifica il valore del contenuto e passa alla locazione successiva.
V - Converte un numero decimale in esadecimale.
J - Torna alla locazione precedente.
N - Ricomincia da una nuova locazione.
Z - Pone uguale a zero il contenuto della locazione.

Dopo aver inserito un valore od aver eseguito i comandi P, V o Z, verranno visualizzati il numero della locazione successiva ed il suo contenuto. Per terminare questo processo, scrivete "end" quando vi viene chiesto il nuovo valore. Sebbene questo programma sia breve e relativamente semplice, esso si dimostrerà molto utile in assenza di un assembler o di un programma commerciale di debug.

Uso di un assembler

L'assembler può essere caricato nella vostra macchina al solito modo. Se possedete entrambe le versioni, a 16K e a 48K, assicuratevi di caricare quella data per la vostra macchina. È importante che non carichiate la copia sbagliata. L'assembler usato per scrivere questo libro viene fornito dalla ACS Software, e si chiama "ultraviolet".

Nel caso decidiate di utilizzare questo stesso assembler, vi sarà utile qualche ulteriore spiegazione sul suo uso.

Per caricare ultraviolet usate semplicemente il comando LOAD (la versione 16K viene caricata in tre parti). Per preparare l'assembler per l'uso premete un qualsiasi tasto: ciò ripulirà la memoria del computer "sotto" il punto in cui viene tenuto l'assembler. Questo viene conservato in una parte di memoria fuori della portata dei programmi BASIC.

Dopo aver fatto queste operazioni vi troverete nel normale modo di comando del BASIC. Potrete a questo punto inserire le istruzioni assembly in un programma proprio come se fosse BASIC. L'unica differenza è che ciascuna di esse dovrà essere preceduta da un'istruzione REM. Ciascuna linea sarà allora così: prima il numero, seguito da REM, ed infine le mnemoniche del linguaggio assembly.

La prima cosa da fare quando si usa questo assembler è decidere esattamente dove si vuol memorizzare il codice ottenuto con l'assemblaggio. Ci sono due semplici alternative: la prima, e probabilmente più utile, delle due è metterlo in memoria da qualche parte al di sopra del programma BASIC ma sotto l'assembler.

NOTA: l'assembler stesso usa, nella versione 48K, gli indirizzi da 60000 in su, ed in quella 16K gli indirizzi da 27500 in su. Non è quindi possibile fare delle operazioni di assemblaggio direttamente in queste locazioni, anche se è possibile aggirare questo problema nel modo che vedremo più avanti.

L'altra alternativa consiste nell'assemblare il codice in un'istruzione REM all'inizio del vostro programma. La si pone all'inizio del programma solo per comodità. Di solito l'indirizzo che si usa per il primo carattere dopo l'istruzione REM della prima

riga è 23760. Se si decide di mettere il codice in un'istruzione REM, cosa questa non consigliabile in quanto si tratta di una complicazione superflua per lo Spectrum, si dovranno lasciare tanti "spazi" o porre altrettanti caratteri dopo la prima istruzione REM quanti sono necessari a contenere il codice. Occorre ricordare cioè che, se si vuole tenere una routine in linguaggio macchina da 25 byte in un'istruzione REM, questa istruzione dovrà avere 25 caratteri dopo la parola REM.

Questo programma assembler riconosce tutte le mnemoniche standard del linguaggio macchina dello Z80 in lettere maiuscole, esattamente come appaiono alla fine del manuale di programmazione BASIC che vi è stato fornito con il computer. Ci sono soltanto un paio di eccezioni a questa regola, che non sono di particolare importanza e che quindi vedremo più avanti. Una cosa importante da notare è che tutti i numeri dovranno essere inseriti in forma decimale, anche se verranno listati in esadecimale dopo essere stati assemblati.

L'inserimento del vostro programma nell'assembler

La prima istruzione di ogni programma che deve essere assemblato è il comando "GO". Non si tratta di una mnemonica dello Z80, ma serve semplicemente per far sapere all'assemblatore che questo è il punto di inizio del programma che deve essere assemblato. La riga successiva deve contenere l'effettivo indirizzo a cui si dovrà collocare il codice operativo. Ciò viene fatto semplicemente con il comando "ORG" seguito dall'indirizzo. Anche ORG, abbreviazione di "Origin" non è una mnemonica dello Z80; serve solo per dire al programma dove mettere il codice assemblato. Se uno di questi due comandi manca in un programma, si avrà una segnalazione di errore quando si tenterà di assemblarlo. Adesso siete pronti per inserire le mnemoniche del vostro programma. Potete mettere più di una mnemonica per riga, purché esse siano separate da un punto e virgola; probabilmente, però, tutto risulterà più facile e più chiaro se terrete una sola istruzione per riga. Dunque adesso abbiamo lasciato un po' di spazio all'inizio del programma per il codice operativo (solo se vogliamo metterlo in un'istruzione REM); abbiamo detto al computer con il comando GO dove inizia il programma da assemblare, e con il comando ORG in che punto della memoria desideriamo mettere il codice assemblato. Adesso possiamo inserire il programma in linguaggio assembler. Spesso, nel corso del programma, può essere utile inserire qualche annotazione o spiegazione di ciò che si sta facendo. Ciò può essere fatto anche usando un assembler, ponendo un punto esclamativo dopo la parola REM e scrivendo quindi il commento. Il punto esclamativo viene usato per dire all'assembler che i codici con le parole che seguono non devono essere assemblate, ma sono lì solo per comodità del programmatore. Se accidentalmente vi dimenticate di mettere il punto esclamativo, l'assemblatore cercherà di convertire in linguaggio macchina tutto quello che avete scritto, e ciò probabilmente farà arrestare l'assembler, con una segnalazione di errore.

Potete adesso scrivere il programma da assemblare. Ciò deve essere fatto a lettere maiuscole, in modo che il computer possa capire cosa scrivete.

NOTA: In un'istruzione di commento REM si possono usare sia lettere maiuscole che minuscole, ma in un'istruzione da assemblare si possono usare solo lettere minuscole.

Quando avrete scritto tutte le mnemoniche in linguaggio assembler che devono essere assemblate, dovrete mettere la parola "Finish" dopo l'istruzione REM. Questa parola dice all'assembler che siete arrivati alla fine della routine che deve essere convertita. Anche "Finish" non è una mnemonica dello Z80.

A questo punto dovreste avere un programma in linguaggio macchina completo, scritto in forma di linguaggio assembler, pronto per essere tradotto in linguaggio macchina. Prima di fare ciò, è bene controllare che non siano stati fatti errori nell'immissione del programma. Ovviamente, se ce ne sono, è molto semplice correggerli con le normali funzioni di editor, proprio come si fa con il BASIC. Per dire al programma assembler di iniziare ad assemblare le mnemoniche già scritte, date RANDOMIZE USR 60000 se avete uno Spectrum 48K oppure RANDOMIZE USR 27500 se avete uno Spectrum 16K. A questo punto compariranno su video le mnemoniche insieme ai relativi codici in linguaggio macchina. Se ci sono errori, cioè istruzioni che l'assembler non capisce, verrà visualizzato un codice di errore e vi sarà chiesto una correzione. Notate che l'assembler non ricontrolla tutto il programma per verificare l'esattezza della vostra correzione. Una cosa interessante è che, quando il programma sta assemblando, la lista del programma in linguaggio macchina viene visualizzata due volte, poiché questo è un assemblatore in due passaggi. Ciò significa semplicemente che esso legge tutto il programma due volte prima di scriverne in memoria la versione finale. I motivi di questo procedimento saranno chiari più tardi.

Adesso sappiamo esattamente ciò che dobbiamo fare in teoria, vediamo se riusciamo a metterlo in pratica. La prima cosa da ricordare è che vogliamo mettere il codice macchina non in un'istruzione REM all'inizio del programma ma, diciamo, all'indirizzo 30000. Il programma che adesso assembleremo viene mostrato qui di seguito; si tratta di una routine per lo spostamento verso destra dei pixel.

```
                          ld hl,22527
                          push hl
                          ld de,23328
                          ld bc,32
                          lddr
                          pop hl
HL=BOT. ADDRESS OF BOT.BLOCK
B=NO.of blocks
                          ld b,3
```

```
A  ---
        push bc
        ld b,8
B     _
        push bc
        push hl
        ld b,8
C  --- --
        push bc
        push hl


        push hl
        pop de
        dec h
        ld bc,32
        lddr
        pop hl
        dec h
        pop bc
        djnz,C
        inc h
        ld de,1760
        push hl
        add hl,de
        pop de
        ld bc,32
        lddr
        pop hl
        ld de,32
        sbc hl,de
        pop bc


        djnz,B
        ld de,1792
        sbc hl,de
        push hl
        ld de,32
        add hl,de
        push hl
        pop de
        pop hl
        push hl
        ld bc,32
        lddr
        pop hl
        pop bc
        djnz,A
        ld de,32
        add hl,de
        ex de,hl
        ld hl,23328
        ld b,32
```

```
D
                ld a,(hl)
                nop
ld a,0 for no wrap
                ld (de),a
                dec de
                dec hl
                djnz,D
                ret
```

Questo programma viene presentato in linguaggio assembly, e sarà compito dell'assembler convertirlo in codici di linguaggio macchina. La prima operazione da fare è caricare l'assembler: questo viene fatto scrivendo LOAD ed assicurandoci di caricare la corretta versione (16K o 48K) per le dimensioni del nostro computer. L'assembler richiederà, per essere caricato, circa un minuto. Se usate l'assembler Ultraviolet, comparirà una grande schermata introduttiva, che scomparirà quando premerete il primo tasto, lasciando sullo schermo il normale messaggio di accensione, cioè: "‹c› 1982 Sinclair Research Ltd". A questo punto siete pronti per inserire le istruzioni in linguaggio assembler tramite un programma BASIC. Potete considerare questa operazione come la scrittura di un normale programma in BASIC. È molto importante ricordare che tutte le istruzioni in linguaggio assembly devono essere precedute da REM. In questo modo la macchina non interpreterà queste linee come istruzioni BASIC.

Adesso bisogna dire all'assembler che c'è un programma da assemblare. Per far questo inseriamo l'istruzione GO come prima riga del programma; ad esempio: 5 REM GO. Dobbiamo quindi specificare in quale punto della memoria desideriamo mettere il linguaggio macchina assemblato. Per questo va usata l'istruzione ORG seguita dall'indirizzo in cui vogliamo che si trovi il primo byte della routine assemblata; il resto della routine verrà poi memorizzato in successione dopo questo indirizzo. Così, per esempio, se la nostra routine è lunga 10 byte e diciamo al computer di iniziare ad assemblare dalla locazione 28000, la routine risiederà nelle locazioni da 28000 a 28900 (in tutto 10 byte). Dato che vogliamo far assemblare il programma visto sopra dalla locazione 30000 in su, la successiva riga del programma BASIC è: 8 REM ORG 30000.

Adesso siamo in pratica quasi pronti per iniziare a scrivere le istruzioni in linguaggio assembly, ma prima di fare ciò può essere utile inserire un commento all'inizio del programma, in modo da poterlo riconoscere facilmente quando lo riprenderemo in mano dopo un certo periodo di tempo. Il modo per inserire un commento nel programma è di mettere un punto esclamativo dopo il numero di linea e l'istruzione REM, seguito dal testo desiderato. Per il programma che stiamo considerando sarebbe sufficiente ricordare che si tratta di un programma per lo spostamento a destra dei pixel. Basterà quindi scrivere: 12 REM! spostamento a destra dei pixel.

Tutto è pronto a questo punto per l'inserimento delle istruzioni assembly. Ricordatevi di scrivere i numeri di linea in tutta la routine, proprio come in un programma BASIC; dovete inoltre anteporre ad ogni comando l'istruzione REM. L'ultimo comando che dobbiamo dare all'assembler è quello con cui diciamo di aver finito la routine da assemblare: ciò viene fatto usando il comando finish. Esso può essere inserito in una linea proprio come abbiamo fatto prima: numero di linea REM finish. Dopo aver inserito tutto ciò nel computer si può iniziare l'assemblaggio.

Se per qualche motivo avete avuto qualche problema nello scrivere queste istruzioni e volete controllare quello che avete scritto, eccovi di seguito la lista di come dovrebbe apparire alla fine il vostro programma BASIC. Non preoccupatevi se i vostri numeri di linea non corrispondono ai miei, ma assicuratevi semplicemente che le effettive istruzioni siano nel giusto ordine.

```
   1 REM go
   2 REM org 30000
  10 REM ld hl,22527
  20 REM push hl
  30 REM ld de,23328
  40 REM ld bc,32;lddr;pop hl;!H
L=INDM BASSO DEL BLOCCO BASSO
  50 REM !B=N. BLOCCHI;ld b,3;A;
push bc;ld b,8;B;push bc;push hl
;ld b,8;C;push bc;push hl;push h
l;pop de;dec h;ld bc,32;lddr;Pop
 hl;dec h;pop bc;djnz,C;inc h;ld
 de,1760;push hl;add hl,de;pop d
e;ld bc,32;lddr;pop hl;ld de,32;
sbc hl,de;pop bc;djnz,B;ld de,17
92;sbc hl,de;push hl;ld de,32;ad
d hl,de;push hl;pop de;pop hl;pu
sh hl;ld bc,32;lddr;pop hl;pop b
c;djnz,A
  60 REM ld de,32;add hl,de;ex d
e,hl;ld hl,23328;ld b,32;D;ld a,
(hl);nop;!ld a,0 for no wrap;ld
(de),a;dec de;dec hl;djnz,D;ret
1000 REM finish
```

Per effettuare l'assemblaggio si accede ad una routine in linguaggio macchina che è già stata caricata nel computer. Dato che esistono due versioni di questo programma, ci sono due diversi indirizzi in cui viene tenuta la routine di assemblaggio. Se avete uno Spectrum 16K date il comando:

RANDOMIZE USR 27500.

Se possedete uno Spectrum 48K dovete invece dare:

RANDOMIZE USR 60000.

IMPORTANTE: Ricordatevi che queste istruzioni per l'assemblaggio, ed anche il modo in cui immettiamo la nostra routine in linguaggio assembly per farla poi leggere dall'assembler si usa solo lavorando con l'assembler dell'ACS. Al momento esistono numerosi tipi di programmi assembler e, probabilmente ce ne saranno ancora di più quando questo libro sarà messo in vendita; mi è quindi impossibile descrivere il modo di funzionamento di ciascuno di essi, dato che spesso le differenze fra i vari programmi sono enormi. Ho preferito invece concentrarmi su di un particolare assembler, pensando che sia uno dei più semplici da usare. Se possedete già un assembler, oppure decidete di comprarne uno diverso dall'ACS, sarà necessario che ne leggiate attentamente il manuale di istruzioni. Questo significa anche che probabilmente l'ultimo listato mostrato non sarà utilizzabile. Naturalmente, la routine che è stata listata un po' prima funzionerà esattamente allo stesso modo, indipendentemente dal tipo di assembler.
Torniamo adesso al nostro compito. Abbiamo preparato il programma BASIC e sappiamo come ordinare al programma assembler di iniziare ad assemblare le istruzioni che abbiamo scritto: allora, procediamo? Dopo aver eseguito il comando per l'assemblaggio indicato sopra, un disegno multicolore apparirà sul video (se usate un televisore bianco e nero vedrete soltanto diverse tonalità di grigio): l'assemblaggio non è ancora completo. Premendo il tasto P un listato della vostra routine verrà inviato alla stampante. Volendo interrompere l'assemblaggio premete invece lo spazio. La pressione di qualsiasi altro tasto farà proseguire l'assemblaggio. È essenziale ricordarsi che è essenziale ricordarsi che si può usare il comando P per la stampante solamente quando la routine viene mostrata per la seconda volta, cioè se ci si trova al secondo "passo" dell'assemblaggio. Premendo poi il tasto ENTER, se compare il messaggio "No Error" alla base del video, la routine in lingaggio macchina è pronta per essere eseguita a partire dall'indirizzo 30000, con i soliti comandi (RANDOMIZE USR 3000 oppure PRINT USR 30000).

Un errore nel vostro programma darà luogo ad una serie di diverse reazioni. Se il difetto risiede nelle istruzioni go, finish e org, una segnalazione di errore verrà data prima dell'inizio dell'assemblaggio.

Se viene trovato un errore durante l'assemblaggio, l'assembler si arresterà con uno di due possibili messaggi di errore. Il primo è un messaggio di errore che compare a intermittenza indicando il numero di riga, il numero dell'istruzione e il tipo di istruzione in cui è stato trovato l'errore: in questo modo è molto facile tornare al programma BASIC per trovare dove si è verificato l'errore. Il secondo messaggio di errore è uno dei messaggi Sinclair. Ci sono tre possibilità. Se avete dato un numero sbagliato, cioè avete cercato di caricare un numero maggiore di 255 in un registro singolo oppure maggiore di 65535 in una coppia di registri,

l'assembler sottrae ripetutamente 256 oppure 65536 da quel numero finché non trova un numero sensato che possa essere usato. Sfortunatamente questo non può venire se il numero troppo elevato si riferisce ad un salto relativo. Sebbene non ci siamo ancora occupati dei salti relativi, cosa che faremo tra breve, questa è una cosa importante da tenere a mente. Nel caso in cui l'assembler non riesca a trovare un valore sensato per il numero, come ad esempio nel caso di un salto relativo, comparirà il messaggio "B Integer out of Range".

Ci sono anche altri casi in cui vengono segnalate condizioni di errore, ma per il momento non sono importanti.

Ora che abbiamo in memoria una routine in linguaggio macchina, è bene che la registriamo su nastro; in questo modo se avverranno degli errori durante l'esecuzione potremo semplicemente ricaricarla da cassetta. L'operazione di copiatura di una routine in linguaggio macchina è ben spiegata nei manuali dello Spectrum, ma la descriverò brevemente.

Tutto ciò che si deve fare è scrivere SAVE "nome" CODE 300000,100. In questo modo si dice al computer di "salvare" cioè di copiare su nastro il programma linguaggio macchina (esso capisce che si tratta di linguaggio macchina perché trova la parola CODE seguita da due numeri) che si trova memorizzato dalla locazione 30000 alla locazione 30000+100, assegnandogli il nome che è stato scritto tra virgolette. Per esempio, se volessimo "salvare" la precedente routine in linguaggio macchina sotto il nome di "pixel-d" dovremmo scrivere il seguente comando: SAVZ "pixel-d" CODE 30000,100. Non dimenticate che stiamo ricopiando su nastro soltanto il programma in linguaggio macchina, e non anche l'assembler o il programma stesso. Per ricaricare in memoria il programma si usa un comando ancora più semplice: LOAD "pixel-d" CODE. Così si caricherà il programma in linguaggio macchina chiamato pixel-d negli indirizzi da cui era stato copiato su nastro, quando il caricamento sarà completo comparirà alla base dello schermo il messaggio "0 OK".

Disassemblatori

Spesso è utile poter effettuare l'operazione esattamente inversa a quella che abbiamo fatto finora. In altre parole invece di assemblare una lista di mnemoniche assembly in linguaggio macchina, possiamo aver bisogno di convertire il codice del linguaggio macchina in mnemoniche assembler. Questo succede spesso quando abbiamo in memoria una routine non scritta da noi, oppure una routine che abbiamo scritto noi ma di cui vogliamo controllare l'esattezza. Un altro vantaggio di molti disassemblatori è che essi mostrano non soltanto le mnemoniche assembler, ma anche i rispettivi codici esadecimali del linguaggio macchina.

Esistono diversi usi di tale possibilità, di cui alcuni non risulteranno per ora molto evidenti. Ho usato estesamente il disassemblatore per produrre i listati nei programmi inseriti in questo libro. L'uso di un disassemblatore è molto più semplice di quello di un assemblatore, e quindi intendo spendere su questo argomento soltanto poche parole.
Per caricare il disassemblatore della "ACS Software" vanno eseguite le stesse operazioni effettuate nel caso dell'assemblatore; allo stesso modo ci si deve ricordare di caricare la versione 16K o quella a 48K a seconda del tipo di computer. È importante anche ricordare che è possibile con entrambe le versioni tenere in memoria sia l'assemblatore che il disassemblatore contemporaneamente. In questo modo si ottiene uno strumento di programmazione molto potente, persino con il semplice Spectrum a 16K. Ciò funziona ricordando che si deve caricare l'assembler prima di disassembler. Quando il disassembler è caricato nel computer, si può far eseguire la routine di disassemblaggio con uno dei seguenti comandi:

Spectrum 48K: RANDOMIZE USR 54000
Spectrum 16K: RANDOMIZE USR 26600

Non appena eseguito uno di questi due comandi compariranno in alto sullo schermo le parole "STARTING ADDRESS?" (indirizzo iniziale?). A questo punto è semplice inserire l'indirizzo, in forma decimale, da cui si vuole che inizi la procedura di disassemblaggio. Se fate un errore mentre scrivete questo indirizzo non usate il tasto DELETE, premete invece il tasto E. Dopo aver correttamente inserito l'indirizzo iniziale, la prima pagina di codice in linguaggio macchina disassemblato comparirà sullo schermo. Se volete continuare il disassemblaggio premete il tasto C; se volete tornare alla fase "starting address?" battete R; se volete stampare una copia del contenuto del video battete P; se infine volete uscire dal disassemblatore e tornare al BASIC battete E.

Una delle caretteristiche più utili dell'assembler come del disassembler è che essi possono rimanere in memoria mentre state operando con un vostro programma in BASIC od in linguaggio macchina, purché il programma in linguaggio macchina non risieda agli stessi indirizzi dell'assembler o del disassembler; questi programmi inoltre non verranno deteriorati dal comando NEW. Come mai, se il comando NEW ripulisce la memoria completamente? Fortunatamente ciò non è vero, dato che, come forse ricorderete, il comando NEW azzera tutti gli indirizzi di memoria dall'inizio della RAM fino alla RAMTOP. Ciò serve appunto per consentire di memorizzare con sicurezza cose come i simboli grafici definibile dall'utente oppure la routine in linguaggio macchina, senza il pericolo di alterazioni per effetto di un programma BASIC o del comando NEW.
Come si può fissare questa RAMTOP ad un certo indirizzo? Si usa il semplice comando CLEAR xxxxx (dove xxxxx indica la locazione della RAMTOP). In questo modo tutta la memoria dopo la RAMTOP risulta protetta. Per esempio,

per proteggere la memoria dalla locazione 28000 in su, in modo che il nostro programma di rinumerazione, più l'assembler e il disassembler possano stare in memoria senza pericolo di alterazioni, useremo il comando CLEAR 27999. È necessario fissare sempre la RAMTOP ad un indirizzo minore di uno di quello da cui vogliamo che inizi l'area protetta.

Troverete che molti programmi hanno l'istruzione CLEAR nel loro listato in BASIC, in modo da assicurare questa protezione. È interessante inoltre notare che se si esce dal disassembler, nel breve programma BASIC in memoria si trova un'istruzione CLEAR. Tenete infine presente che anche se CLEAR ripulisce il video, non ripulisce la memoria sopra la RAMTOP.

Break in linguaggio macchina

Il modo in cui funziona in BASIC il tasto BREAK è molto semplice. Il computer controlla la tastiera alla fine di ciascuna istruzione, e se il tasto BREAK è stato premuto interrompe immediatamente l'esecuzione, fa comparire sul video un messaggio e ritorna al modo comando. Sfortunatamente quando ci troviamo in una routine in linguaggio macchina non possiamo ottenere questo risultato in modo diretto, poiché questa possibilità non esiste a livello di hardware della macchina. Possiamo però aggirare il problema preparando una nostra breve routine di BREAK, che può essere memorizzata insieme alle altre routine in linguaggio macchina per essere richiamata ogni tanto a controllare se c'è stata pressione del tasto BREAK. Questa operazione viene eseguita dalla routine seguente. Non tutte le istruzioni presenti in questo programma sono già state spiegate, ed in effetti alcune di esse, in particolare l'istruzione RRA, non verranno menzionate affatto. Basti sapere che se si preme il tasto BREAK questo programma provocherà un immediato ritorno al BASIC.

```
ld a,127
in a,(254)
rra
ret nc
```

Per il momento possiamo soltanto mettere questa routine all'interno dei nostri programmi in linguaggio macchina, ma più avanti impareremo ad usarla come una subroutine dei nostri programmi principali. Otterremo così uno strumento molto utile e potente. Più avanti vedremo anche come controllare se sia stato premuto un qualsiasi altro tasto predeterminato.

Monitor

La domanda che senza dubbio vi sarete già posti è cosa facciamo se non abbiamo un assembler, se non lo vogliamo comprare o semplicemente non ce

lo possiamo permettere? Non esiste un modo semplice di ottenere tutti i vantaggi di un assembler senza acquistare un tale programma, ma si possono lo stesso inserire in memoria i codici del linguaggio macchina con alcuni dei vantaggi offerti da un assembler; per questo occorre usare un monitor. Ci sono anche alcuni vantaggi che non si otterrebbero con un assembler e che compensano un poco del fatto che si deve effettuare "a mano" la conversione delle mnemoniche assembly in codici operativi del linguaggio macchina, usando le tabelle di conversione riportate alla fine di questo libro. Queste tabelle sono disponibili anche da altre fonti.

La correzione dei programmi

Presto vi renderete conto se pure avete controllato meticolosamente una certa routine in linguaggio macchina, è facile che ogni tanto si verifichino degli errori. Si infilano dappertutto, e non possiamo farci niente! Non è possibile rimediare in alcun modo ad un blocco nell'esecuzione di un programma in linguaggio macchina, ed è quindi particolarmente importante cercare di trovare gli errori della sua effettiva esecuzione. Per questo motivo, e per altri che vi saranno chiari in seguito, è veramente un gran vantaggio poter disporre di un programma "debugger". Questo tipo di programma può essere acquistato già pronto, in "packages", proprio come un assembler o un disassembler. Ancora una volta, per dovere di informazione, devo dirvi che il programma di correzione degli errori che ho usato in questo libro e che userò come esempio specifico sia ora che in seguito, non è stato prodotto dalla stessa ditta di software da cui provengono il mio assembler e il mio disassembler, anche se probabilmente essa ne avrà uno disponibile al momento della pubblicazione di questo libro. Il debugger che ho usato qui è fornito dalla "ARTIC Computing".

Il caricamento di questo programma debugger nello Spectrum è semplicissimo: basta scrivere LOAD "", e ricordarsi di caricare la versione adatta alla macchina che si sta adoperando. Una volta che si sia caricato il debugger, questo potrà operare in diversi modi. Tuttavia, poiché questo programma è stato ideato per essere usato con le vostre routine in linguaggio macchina, è ragionevole che prima si debba caricare una di tali routine, per esempio quella di spostamento dei pixel che abbiamo memorizzato precedentemente in questo capitolo. Sebbene in questo programma non ci siano errori (li abbiamo già eliminati!), potrà servire lo stesso come utile esempio. Inoltre, poiché non è possibile caricare direttamente le proprie routine che non siano state conservate su nastro tramite il debugger stesso, occorre anzitutto uscire dal programma debugger e tornare al BASIC per effettuare il caricamento. Per fare ciò si deve premere il tasto "X", seguito da ENTER. Dopo aver fatto ciò potete caricare normalmente la vostra routine in linguaggio macchina. Per tornare al programma debugger del linguaggio macchina dovete scrivere "PRINT USR 30884" per la versione 16K, oppure "PRINT USR 63652" per la versione 48K. È importante che scriviate correttamente questi numeri (sono riportati anche nel manuale di istruzioni che accompagna il

debugger). Tutte le funzioni svolte dal debugger sono elencate nelle istruzioni, senza però una spiegazione completa, a puro scopo di riferimento. Non intendo qui prendere in esame tutti i comandi e le funzioni disponibili in questo debugger; vediamo invece brevemente quelle più importanti:
Il comando "Z" consente il disassemblaggio di una piccola zona di RAM, proprio come faceva il disassembler di cui abbiamo già parlato, anche se il disassembler contenuto nel debugger non è molto sofisticato. Per disassemblare l'area in cui si trova la vostra routine di spostamento dei pixel basta scrivere: Z 7530. Se per qualche motivo volete interrompere il disassemblaggio, premete il tasto BREAK. Ricordatevi sempre di premere ENTER dopo aver scritto ogni istruzione per il debugger: è facile dimenticarsene.

NOTA: Tutti i numeri usati in questo programma debugger sono esadecimali, il che significa che ci sono al massimo quattro cifre; i numeri decimali non sono ammessi. Per questo motivo può essere utile tornare a vedere i precedenti capitoli per ripassare il modo in cui si effettuano le conversioni da decimale a esadecimale. Per venirvi incontro, comunque, alla fine di questo libro, nell'appendice A, si trova una lista completa dei numeri binari, decimali ed esadecimali da 0 a 255, che può essere molto utile nelle conversioni.
Per eseguire, dall'interno del programma debugger, una routine in linguaggio macchina, si usa il comando G seguito dall'indirizzo, in esadecimale, da cui inizia quella routine, seguito come sempre da ENTER. Si può poi mostrare il contenuto dei principali registri premendo D e poi ENTER: in questo modo in alto sullo schermo compariranno i valori contenuti nei registri principali.

Quest'ultimo comando è molto utile alla fine di un programma, ma non sarebbe ancora più utile poter conoscere il contenuto dei vari registri a metà strada nell'esecuzione di un programma? Ciò può essere facilmente ottenuto mediante l'utilizzo dei cosiddetti "breakpoints", cioè punti di arresto (detti anche "quit points"). Essi non sono altro che degli indirizzi che, raggiunti dal programma durante l'esecuzione, provocano un ritorno immediato al debugger. Potrete allora farvi mostrare il contenuto dei registri e, come vedrete più avanti, anche lo stato dei vari segnali ("flags"). Per fissare un "breakpoint" si usa il comando Q seguito dall'indirizzo, in esadecimale, in cui si vuole porre l'arresto. Poi si fa eseguire la routine in linguaggio macchina con il comando G che abbiamo già spiegato. Non appena l'esecuzione della routine si arresta e si ritorna al debugger, il "breakpoint" viene cancellato.
Adesso provate a disassemblare il programma, decidete dove mettere i "breakpoints", inseriteli, eseguite il programma e quindi fatevi mostrare il contenuto dei registri principali dopo ciascun punto di arresto. Inoltre, usando il comando F, seguito da ENTER, potrete vedere sullo schermo il contenuto dei segnali. Esistono anche altri comandi meno spettacolari all'interno del programma debugger della Artic, che adesso descriverò brevemente. Le operazioni possibili sono:

1. Inserire un messaggio in una certa porzione di memoria, scrivendo semplicemente il messaggio tramite la tastiera, che poi lo convertirà negli opportuni codici esadecimali.

2. Cercare un certo valore all'interno di un dato blocco di memoria, e vedere poi l'indirizzo in cui esso è stato trovato.

3. Copiare un blocco di memoria in un altro.

4. Mostrare il contenuto dei registri alternativi, che dobbiamo ancora considerare.

5. Ordinare al debugger di sostituire tutti i numeri che hanno un certo valore con altri.

6. Cercare e scaricare dal nastro ii che sono stati scritti per mezzo del debugger stesso.

7. Modificare il contenuto dei vari byte di memoria, ed inserire nei vari indirizzi codici del linguaggio macchina per poi farli eseguire.

8. Assegnare particolari valori a certi registri, in modo da poter simulare determinati eventi nell'ambito del programma.

9. Stampare i codici dei caratteri BASIC corrispondenti ai codici di una routine, così da poterli inserire in una istruzione REM.

Cosa compare?

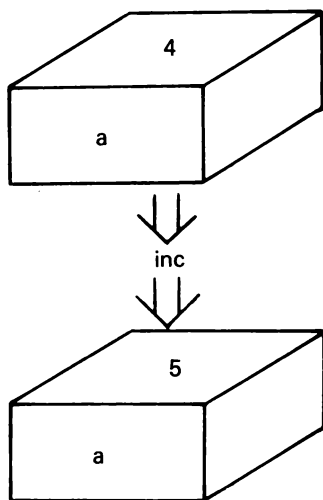
Se siete alle strette e potete permettervi di comprare uno solo di questi programmi, allora probabilmente la cosa migliore è che vi compriate per primo il debugger. Questo non solo perché esso è di considerevole aiuto quando si devono cercare gli errori in un programma, ma anche perché vi può aiutare moltissimo nello scrivere i programmi e nel comprendere come funzionano. Se invece decidete di approfondire maggiormente le vostre conoscenze di linguaggio macchina e volete mettervi a scrivere programmi più lunghi e complicati, è facile che l'assembler risulti lo strumento più adatto, purché sia accoppiato ad un disassembler. Poi ci vorrà anche un debugger.

E finalmente...

Prima di continuare con i nostri programmi, cosa che faremo nel prossimo capitolo, spiegherò velocemente quattro utilissime istruzioni che sono già state usate senza che voi forse le abbiate comprese appieno, essendo state solo accennate. Come per tutte le cose che già abbiamo visto, anche queste saranno più chiare se messe in pratica. Per questo motivo il prossimo capitolo è interamente dedicato alla stesura di programmi in linguaggio macchina. Si porrà anche grande attenzione nel descrivere attentamente ciascuna istruzione e tutte le

procedure necessarie per scrivere un buon programma. Le istruzioni che vedremo sono: INC, DEC, NOP, RET.

INC - È un'altra di quelle istruzioni che possono essere usate in due forme, cioè su un registro singolo oppure su una coppia di registri. Uno dei principali vantaggi di questo comando è che esso può essere usato con tutti i registri. INC x farà sì che il valore contenuto nel registro x venga incrementato di 1. Per esempio, se il registro A contiene il valore 5, con l'esecuzione del comando INC A il valore di questo registro diventerà 6. L'effetto sarà esattamente identico anche se applicato agli altri registri. Inoltre, se il valore di un qualsiasi registro è 255 ed esso viene aumentato di 1 con il comando INC, il suo valore tornerà a zero. Ma ecco la novità: se il valore di un registro passa da 255 a 0 con il comando INC, il segnale di riporto non verrà posto uguale a 1 come succederebbe con l'istruzione ADD. Questo è molto importante da ricordare quando, in seguito, si vorrà controllare il valore di un registro per decidere sulla successiva direzione che il programma dovrà prendere.

Il disegno che segue è una semplice rappresentazione del modo di funzionamento di questa istruzione.

Se si vuole aumentare di 1 il contenuto di una variabile in BASIC, si usa l'istruzione LET B=B+1, qualora la variabile interessata sia B. È evidente da questo esempio che l'equivalente in linguaggio macchina è decisamente più breve, facendi quindi

risparmiare memoria, e senz'altro molto più rapido. Le mnemoniche assembly e gli equivalenti codici esadecimali sono indicati qui di seguito:

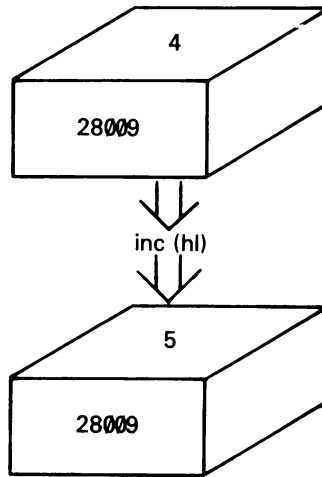
```
INC A    3C
INC B    Ø4
INC C    ØC
INC D    14
INC E    1C
INC H    24
INC L    2C
```

È anche possibile incrementare di 1 il valore contenuto in una coppia di registri, proprio come per i registri singoli. L'unica differenza è che l'istruzione INC non modificherà in alcun modo i vari segnali, incluso quello di riporto.
Le mnemoniche assembly ed i codici esadecimali per queste istruzioni sono:

```
INC BC   Ø3
INC DE   13
INC HL   23
```

Il funzionamento di queste istruzioni non è difficile da capire, anche se la cosa può non essere del tutto immediata. Se per caso doveste avere dei problemi, eccovi un programma in linguaggio macchina per illustrarne il funzionamento. Eseguendo questo programma dal BASIC (con l'istruzione PRINT USR 30000), il valore finale della coppia BC verrà mostrata su video. È interessante giocare con il contenuto di BC nella prima istruzione, in modo da vedere come ne risulterà influenzata la risposta.

```
org 30000
30000 01 00 00      ld bc,0000
30003 0B            dec bc
30004 C9            ret
```

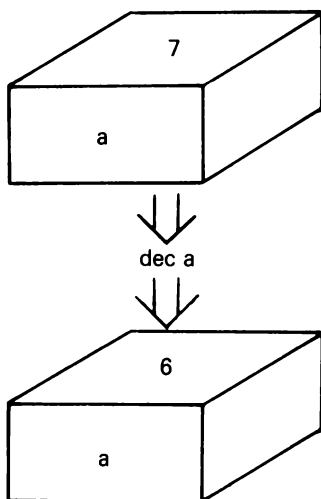
C'è un'altra funzione dell'istruzione INC: è possibile incrementare di 1 il contenuto di una particolare locazione di memoria. Per esempio, se caricate nella coppia HL l'indirizzo della locazione il cui contenuto deve essere incrementato, e poi fate eseguire il comando INC (HL), il contenuto dell'indirizzo HL verrà aumentato di 1, mentre il valore di HL stesso non verrà modificato. Il disegno che segue mostra tutto ciò un po' più chiaramente.

Il codice esadecimale in linguaggio macchina per questa istruzione è 34. Ricordatevi però che questa istruzione può essere usata soltanto in riferimento al registro HL.

Adesso scrivete il programma listato qui sotto, eseguitelo, e controllate il contenuto della locazione 28012. Eseguite di nuovo la routine e controllate il contenuto della stessa locazione: troverete che è aumentato di 1. Fate attenzione che l'operazione di aumentare di 1 il valore di una locazione funziona solo su quella particolare locazione; quindi il valore massimo cui si può giungere è 255, dato che questo è il numero più grande che possa essere memorizzato in una singola locazione, come già ben sapete. Ecco il programma:

```
org 30000
30000 01 00 00      ld bc,0000
30003 03            inc bc
30004 C9            ret
```

DEC:
Questa istruzione è analoga a INC, tranne per il fatto che il valore del registro o della locazione viene diminuito di 1 invece che aumentato. Per esempio, se facciamo seguire il comando DEC A quando A ha un valore pari a 7, il risultato sarà che 1 viene sottratto al valore del registro A, ottenendo alla fine 6. Il seguente disegno mostra chiaramente questo processo.

Ecco ora la lista delle mnemoniche assembly del linguaggio macchina per queste istruzioni:

DEC A	3D
DEC B	Ø5
DEC C	ØD
DEC D	15
DEC E	1D
DEC H	25
DEC L	2D

Dovreste ora essere in grado di convertire il programma di esempio usato prima per mostrare il funzionamento sia di INC che di DEC. Se ancora fate un po' di confusione, ecco la routine opportunamente modificata:

```
org 30000
30000 21 6C 6D        ld hl,28012
30003 35              dec (hl)
30004 C9              ret
```

Anche qui il segnale di riporto non viene toccato quando si fa eseguire l'istruzione di decremento sia su un registro singolo che su una coppia di registri. Le

mnemoniche assembly ed i codici esadecimali del linguaggio macchina per le istruzioni di decremento con coppie di registri sono:

```
DEC BC    ØB
DEC DE    1B
DEC HL    2B
```

Come potete vedere da questi codici e mnemoniche, i codici esadecimali hanno in effetti qualche relazione con le mnemoniche assembler. Per esempio, tutte le istruzioni di decremento su coppie di registri hanno B come seconda cifra. È utile ricordare piccole informazioni come questa: facilitano la vita se non si possiede un assembler.

Sospensete un momento la lettura e tornate indietro per vedere se trovate altri casi di simili relazioni.

Come per l'istruzione di aumento, è possibile diminuire di 1 anche il contenuto di una locazione particolare, quando l'indirizzo di questa locazione viene tenuto nella coppia di registri HL. Ecco un esempio di questo tipo di istruzione:

```
org 30000
30000 21 6C 6D      ld hl,28012
30003 34            inc (hl)
30004 C9            ret
```

Dopo aver eseguito questa routine, controllate il valore della locazione 28012 con il comando "PEEK" del BASIC; eseguite di nuovo la routine e controllate ancora una volta il contenuto della stessa locazione. In tutti i casi, tranne se il valore precedente di quella locazione era 0, il contenuto risulterà diminuito di 1. Se però il contenuto della locazione era già 0, esso diverrà 255.

RET:
Ormai avrete certamente capito che questa istruzione fa sì che termini l'esecuzione della routine, e che il controllo del computer ritorni al BASIC. L'istruzione "RET" alla fine di un programma in linguaggio macchina è sempre necessaria, a meno che non si faccia uso di una delle varianti particolari di cui ci occuperemo più avanti, poiché per ora non sono importanti. Il codice esadecimale corrispondente alla mnemonica "RET" dell'assembly è facile da ricordare, e probabilmente tra qualche pagina vi sarà diventato familiare: esso è "C9". Il motivo di tale familiarità è che lo dovrete usare continuamente.

NOP:
Questo comando è molto semplice da capire: esso non fa assolutamente niente. Quando il computer incontra questa istruzione, il cui codice è 0, non fa nulla e passa all'istruzione successiva, lasciando i contenuti di tutti i registri, di tutti gli indirizzi e di tutti i segnali esattamente come si trovavano.
Può sembrare strano che si sia voluta includere un'istruzione che apparentemente non serve a niente: essa è invece molto utile. È spesso importante, in una routine, poter lasciare delle zone nelle quali non c'è nulla: in questo modo successivamente si potranno inserire altri comandi. Questa è una pratica simile a quella secondo cui in BASIC si saltano alcuni numeri nell'assegnare i numeri di riga ad ogni istruzione. In altri termini non è mai conveniente scrivere un programma con le linee numerate 1, 2, 3, 4 e così via, senza lasciarsi alcuna possibilità di manovra. Inoltre è molto utile poter disporre di un comando che assicura che, se necessario, il computer non farà assolutamente nulla.

Finora ci siamo concentrati soprattutto sull'aspetto teorico del linguaggio macchina, anche se ho usato molti esempi ed introdotto alcuni programmi per vostro uso. Tuttavia, l'utilizzo vero e proprio del linguaggio macchina è la sua principale attrattiva. Il capitolo seguente sarà quindi interamente dedicato al processo di costruzione di un'idea di programma, la sua codificazione e infine la correzione degli errori, l'esecuzione e la memorizzazione su nastro. Prima di finire questo capitolo, eccovi un programma complementare a quello visto prima: questo serve a spostare verso l'alto i pixel. Potete usare un assembler, il semplice monitor oppure un debugger.

Buon divertimento!

```
org 30000
30000 21 00 40     ld hl,16384
30003 E5           push hl
30004 11 00 5B     ld de,23296
30007 01 20 00     ld bc,32
30010 ED B0        ldir
30012 E1           pop hl
'HL=IND. ALTO DEL BLOCCO ALTO
'B=N. BLOCCHI
30013 06 03        ld b,3
A
30015 C5           push bc
30016 E5           push hl
30017 06 08        ld b,8
B
30019 C5           push bc
30020 E5           push hl
30021 06 07        ld b,7
C
30023 C5           push bc
```

```
30024  E5              push hl
30025  E5              push hl
30026  D1              pop de
30027  24              inc h
30028  01  20  00      ld bc,32
30031  ED  B0          ldir
30033  E1              pop hl
30034  24              inc h
30035  C1              pop bc
30036  10  F1          djnz C
30038  E5              push hl
30039  11  E0  06      ld de,1760
30042  ED  52          sbc hl,de
30044  D1              pop de
30045  01  20  00      ld bc,32
30048  ED  B0          ldir
30050  E1              pop hl
30051  11  20  00      ld de,32
30054  19              add hl,de
30055  C1              pop bc
30056  10  D9          djnz B

30058  11  E0  06      ld de,1760
30061  19              add hl,de
30062  E5              push hl
30063  11  20  00      ld de,32
30066  19              add hl,de
30067  D1              pop de
30068  01  20  00      ld bc,32
30071  ED  B0          ldir
30073  E1              pop hl
30074  11  00  08      ld de,2048
30077  19              add hl,de
30078  C1              pop bc
30079  10  BE          djnz A
30081  11  20  00      ld de,32
30084  ED  52          sbc hl,de
30086  EB              ex de,hl
30087  21  00  5B      ld hl,23296
30090  06  20          ld b,32
D

30092  7E              ld a,(hl)
30093  00              nop
'LD A,0 PER NON RITORNO
30094  12              ld (de),a
30095  13              inc de
30096  23              inc hl
30097  10  F9          djnz D
30099  C9              ret
```

CAPITOLO 7

LA SCRITTURA DI UN PROGRAMMA

Siamo già andati molto avanti nel cammino per imparare come si usa il linguaggio macchina e le sue varie istruzioni. È ormai tempo che iniziamo a vedere come possiamo mettere insieme un nostro programma in linguaggio macchina. Conoscendo soltanto pochi comandi possiamo fare tutte le seguenti operazioni e quindi utilizzarle per i nostri fini:

1 sommare i registri;
2 sommare numeri ai registri;
3 sottrarre un registro da un altro;
4 sottrarre numeri dai registri;
5 incrementare di uno un registro;
6 incrementare di uno il valore di una locazione;
7 diminuire di uno un registro;
8 diminuire di uno il valore di una locazione;
9 caricare un valore in un registro;
10 caricare in un registro il valore di un altro registro;
11 caricare in un registro il contenuto di una particolare locazione;
12 caricare in una particolare locazione il valore di un registro;
13 ritornare al BASIC senza fare assolutamente nulla.

Con queste basi possiamo preparare delle routine in linguaggio macchina anche molto complesse.
Tuttavia è anche giunto il momento che vi rendiate conto che la programmazione in linguaggio macchina non è così semplice come in BASIC (però non demoralizzatevi e riuscirete ad ottenere buoni risultati): non potete semplicemente sedervi alla tastiera e scrivere. Bisogna invece procedere in modo logico e sistematico; dovete decidere passo per passo cosa volete fare e come volete farlo.
Assicuratevi sempre di non aver fatto errori, dato che è impossibile recuperare una situazione di errore in linguaggio macchina. Un altro fattore importante da ricordare è che un programma in linguaggio macchina non è semplice da rivedere e correggere come un programma in BASIC. Quindi l'utilizzo di "flow-chart" e di

appunti vari su ciò che esattamente si vuole che succeda nel programma può essere di quasi inestimabile aiuto. Ciò che segue è un profilo del procedimento di preparazione e di costruzione di un mio programma. Con un pò di pratica arriverete ad avere un vostro metodo personale per affrontare il problema, ma inizialmente vi consiglio di seguire questo mio procedimento.

1 L'idea in breve

Decidete esattamente cosa volete che il programma o la routine siano in grado di fare una volta terminato il lavoro. Ciò richiede un esame molto attento del problema che si vuole risolvere o delle idee che si vogliono mettere in pratica, e decidere poi esattamente cosa ci si aspetta che il computer faccia. Scrivetevi tutte queste cose e usatele poi come riferimento man mano che andate avanti.

2 Un "flow-chart" schematico

Questo vi assicura di aver spezzato la vostra idea generale nelle sue parti componenti. Tutto vi riuscirà più facile quando potrete lavorare su un passo alla volta.
Fino a questo momento non è necessario scendere ai minimi dettagli di ciò che si vuol eseguire in ogni passo; è importante solo stabilire bene l'ordine con cui dovranno essere eseguite le varie parti della vostra idea.

3 Lo studio della struttura

Ciò richiede l'esame di tutto il programma, passo per passo, analizzando quali movimenti o operazioni il computer deve compier per affrontare ogni singolo punto del lavoro. Tenete presente il tipo di comandi che avete a disposizione. Da qui il passaggio allo stadio succesivo è una progressione del tutto naturale.

4 Il "flow-chart" finale

Questo è in pratica l'amalgama di tutti i singoli passi nel funzionamento della routine, messi in modo sequenziale e con logica in una forma facilmente comprensibile.

5 La conversione

Non si tratta di cambiare il linguaggio assembly in linguaggio macchina, bensì

della semplice trasformazione di ciò che avete scritto sotto forma di brevi frasi di istruzioni di linguaggio assembly. Queste possono poi essere scritte in linguaggio comprensibile per l'assemblatore, cioè usando le mnemoniche. Quindi mettete le istruzioni nel loro ordine logico, proprio come desiderate che il computer le esegua.

6 Il "Dry-run"

Adesso eseguite il programma passo dopo passo, non con il computer ma con carta e penna, scrivendovi il contenuto dei vari registri che usate, il contenuto dei particolari indirizzi che vengono modificati e lo stato dei segnali stessi. Se tutto ciò funziona bene siete pronti per iniziare l'inserimento delle istruzioni nell'assembler, altrimenti avrete almeno risparmiato il tempo necessario per la conversione in linguaggio macchina, per l'inserimento dei codici nel computer e per l'esecuzione prima di scoprire l'errore.

7 L'immissione

È molto importante avere piena familiarità con il metodo di utilizzo del vostro assembler, e questo è il motivo per cui ho scelto di usare sempre lo stesso per tutto il libro.

8 Il controllo

Prima di procedere oltre vale la pena di effettuare un rapido controllo, con il disassemblaggio oppure mediante un completo esame dei comandi, nell'eventualità che abbiate fatto qualche piccolo errore durante l'immissione del programma nell'assembler. Spesso gli errori vengono individuati dall'assembler, ma altrettanto spesso ciò non succede e quindi può venir assemblata un'istruzione del tutto diversa.
In questo caso può succedere che salti tutto il programma senza che risulti chiaramente cosa abbia provocato l'errore.

9 La correzione dell'errore

In questa fase ci si può aiutare con un programma debugger, di cui abbiamo già visto il funzionamento. La correzione di un programma mentre esso si trova in memoria non solo è di notevole aiuto per eliminarne gli errori, ma dà anche la possibilità di capire molto bene cosa succede all'interno del computer.

10 La conservazione su nastro

Come ho già fatto notare la conservazione su nastro di un programma in linguaggio macchina non richiede le stesse operazioni di quella di un programma in BASIC. È importante quindi essere in grado di compiere entrambe le operazioni, in modo da potersi costruire una memoria permanente delle proprie routine.

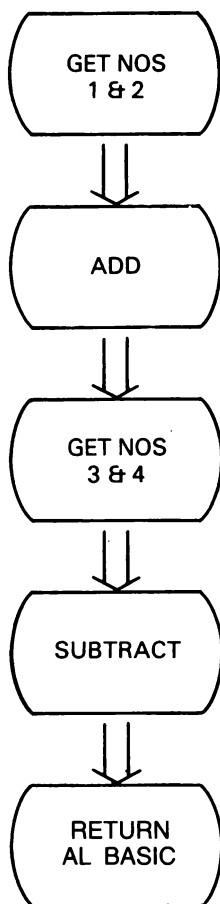
11 L'esecuzione

Siete ora arrivati al punto in cui tutto funziona bene e siete pronti a fare eseguire la vostra routine. Anche questa non è un'operazione del tutto semplice e immediata.
Iniziamo ora questo procedimento con una nostra idea per un programma, in modo da poterlo seguire fino ad ottenere il programma finale per la conclusione di questo capitolo.

L'idea

Ciò che faremo adesso è scrivere una routine in linguaggio macchina con le seguenti caratteristiche:

1 che sia accessibile dal BASIC;
2 che sia in grado di comunicare con il BASIC;
3 che possa sommare due numeri a 16 bit;
4 che possa sottrarre un numero a 16 bit da un altro, lasciando il
risultato disponibile per l'utilizzo da un programma BASIC.

Il flow-chart schematico

Lo studio della struttura

Il primo problema che incontriamo riguarda il modo in cui possiamo assegnare i numeri dal BASIC al programma in linguaggio macchina, cosicché i numeri da 16 bit che devono essere sommati o sottratti siano controllati dal BASIC. Dovete ricordarvi che non è possibile assegnare semplicemente un numero ad una variabile in BASIC ed aspettarci che il linguaggio macchina lo possa riconoscere.

Sarebbe una procedura molto complicata trasferire il contenuto di una variabile del BASIC ad un registro del linguaggio macchina, per cui credo che questa non sia una procedura conveniente. Ricordatevi anche che non è possibile caricare un numero in un registro direttamente tramite il BASIC, dato che nel BASIC non ci sono istruzioni per la manipolazione dei registri. Fortunatamente invece, esiste un'operazione che può essere eseguita sia tramite il BASIC che tramite il linguaggio macchina, e cioè il trasferimento di un numero in una particolare locazione di memoria e poi il trasferimento di quello stesso numero dalla locazione ad un registro oppure ad una variabile del BASIC.

Se vogliamo caricare un numero in una certa locazione tramite il BASIC, usiamo l'istruzione "POKE"; se poi vogliamo trasferirlo dalla locazione ad un registro usiamo l'istruzione "LOAD locazione". Ciò che stiamo facendo in pratica non è altro che mettere un valore in una scatola, passare al linguaggio macchina e poi tirarlo fuori di nuovo.

Il problema successivo è quello relativo al trasferimento delle risposte dal linguaggio macchina ad una forma che sia di nuovo accessibile tramite il BASIC. In questo caso ci sono due diversi metodi per risolvere il problema. Per quanto detto prima, non avrebbe assolutamente senso cercare di accedere ad un registro del linguaggio macchina direttamente dal BASIC.

La prima soluzione possibile è molto simile al modo in cui si effettua il trasferimento nel caso precedente: mentre si è ancora in linguaggio macchina si carica la risposta in una particolare locazione di memoria e poi la si recupera dopo aver fatto ritorno al BASIC.

D'altra parte, se si usa l'istruzione "PRINT USR xxxxx", dove xxxxx è l'indirizzo del programma in linguaggio macchina, quando questo programma ha terminato l'esecuzione il contenuto della coppia di registri BC verrà stampato sul video. Ciò significa che, se facciamo in modo che il risultato del calcolo sia contenuto nella coppia BC, potremo risolvere il nostro problema usando l'istruzione "PRINT USR". Analogamente potremmo usare un'istruzione del tipo "LET A = USR xxxxx", dove xxxxx è nuovamente l'indirizzo del programma in linguaggio macchina: in questo caso il contenuto della coppia di registri BC verrà trasferito alla variabile A. Questa soluzione è veramente molto utile, ma il suo maggiore difetto è che nei registri BC si può memorizzare soltanto una risposta. Allora, piuttosto che ritornare al BASIC e dover poi entrare di nuovo nel linguaggio macchina, è consigliabile usare una combinazione delle due soluzioni. In altre parole memorizziamo una risposta in qualche locazione che sia accessibile dal BASIC e carichiamo l'altra risposta in BC, cosicché essa potrà essere stampata oppure assegnata ad una variabile.

Avendo così deciso come effettueremo il trasferimento delle costanti dal BASIC al linguaggio macchina e le risposte dal linguaggio macchina di nuovo al BASIC,

dobbiamo adesso studiare la sequenza degli eventi che intercorrono fra queste due operazioni. Dobbiamo decidere cioè se vogliamo eseguire prima la routine di addizione oppure prima quella di sottrazione, la scelta è del tutto libera, basta che ci ricordiamo di mettere il risultato della prima routine in una locazione di memoria e quello della seconda routine nella coppia di registri BC. Io ho scelto per il nostro programma di fare prima l'addizione e poi la sottrazione. (Questa decisione non l'ho raggiunta dopo un complicato processo analitico, ho semplicemente fatto a testa o croce con una moneta).

Flow-chart - seconda versione

Ecco ora la seconda versione del flow-chart originario. Esaminatelo attentamente, tenendo presente quanto si è detto finora:

Prendi i primi due valori dalle loro particolari locazioni di memoria.

Somma questi due numeri.

Memorizza il risultato in una scatola, ovvero in una locazione.

Prendi gli altri due valori dalle loro rispettive locazioni di memoria.

Sottrai uno dall'altro.

Memorizza il risultato nella coppia di registri BC.

Ritorna al BASIC.

INVECE, PER LA PARTE IN BASIC:

Memorizza quattro numeri nelle locazioni di memoria prestabilite.

Esegui la routine in linguaggio macchina.

Stampa il risultato della seconda parte della routine.

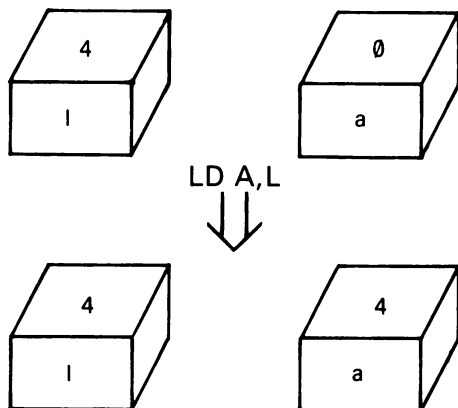
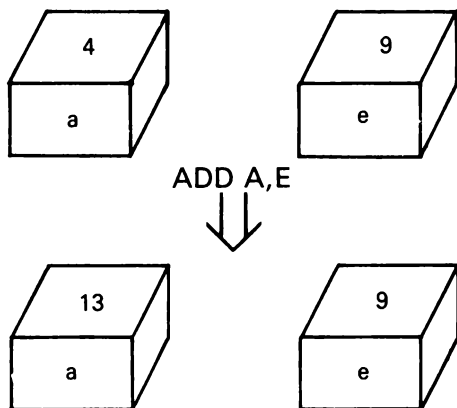
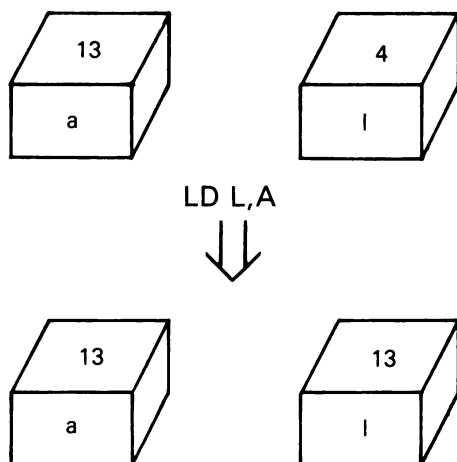
Recupera e stampa sul video il risultato della prima parte della routine, che era stato memorizzato in una certa locazione di memoria nell'ambito del linguaggio macchina.

Stop.

Come sommiamo i due numeri da 16 bit?

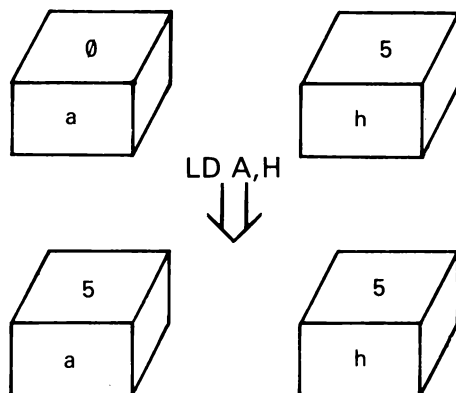
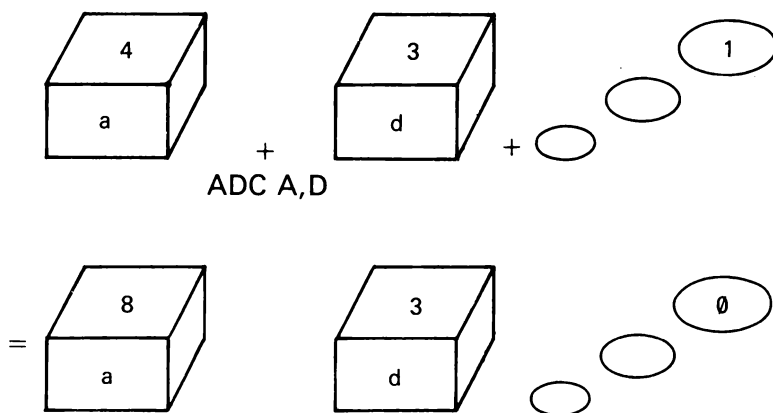
Supponiamo che i numeri presi dalla memoria del computer siano stati caricati nei registri HL e DE. Il risultato dell'addizione verrà poi messo in HL. È importante ricordare che H è il byte più significativo nella coppia HL e L è quello meno significativo, così come D è il byte più significativo nella coppia DE ed E è quello meno significativo. Proprio come nell'addizione ordinaria, si sommano per prime le parti meno significative, fare l'opposto sarebbe oltremodo problematico, come vi renderete conto più avanti.

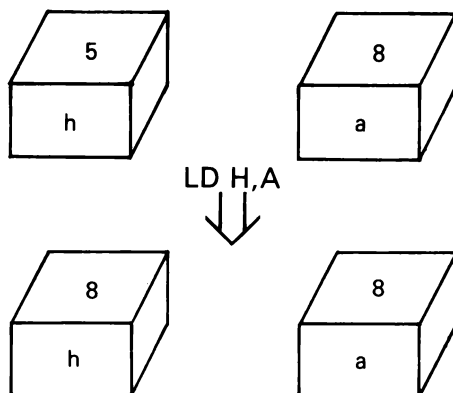
Dobbiamo quindi sommare il registro E al registro L. Sfortunatamente però, come forse ricorderete, un registro può essere sommato soltanto al registro A. E allora come facciamo? È semplice: dobbiamo solo trasferire il contenuto del registro L nel registro A, poi sommare il registro E al registro A. Il risultato della somma di E con L si trova ora nel registro A, cosicché la risposta in A dovrà poi essere trasferita di nuovo in L.

Abbiamo così deciso quali saranno le prime due istruzioni del nostro programma in linguaggio macchina. Anzitutto dobbiamo "trasferire il contenuto di L in A", come indicato nel seguente disegno:

Poi sommiamo il registro E al registro A, come mostrato sotto graficamente:

Adesso abbiamo ottenuto nel registro A il risultato voluto, ma esso non si trova dove volevamo. Noi vogliamo che tutto il risultato sia contenuto nella coppia HL, quindi dobbiamo mettere il risultato della somma dei due byte meno significativi nel registro L. Ciò può essere fatto con un semplice trasferimento del contenuto di A in L.

Dobbiamo adesso considerare brevemente cosa succede se i due byte da sommare portano ad un risultato maggiore di 255, maggiore cioè del più alto numero che può essere contenuto in un registro singolo. Una buona cosa dei

microprocessori è che la risposta in questa circostanza è del tutto corretta e prevedibile: si torna a zero! Sommando 1 a 255 si torna a 0; sommando 20 a 255 si arriva a 19 passando per 0. Inoltre, il segnale di riporto viene posto uguale a 1. Il segnale di riporto è un solo bit del registro F cui la CPU stessa fa riferimento per vedere se un numero è andato fuori scala o no. Se un numero ha superato la barriera di 255, è andato senz'altro fuori scale, e il segnale di riporto viene posto a 1. Ovviamente, se non si va fuori scala il segnale di riporto viene rimesso a 0. È assolutamente necessario ricordare ogni tanto questo fatto, e non si sottolinea mai troppo la sua importanza.

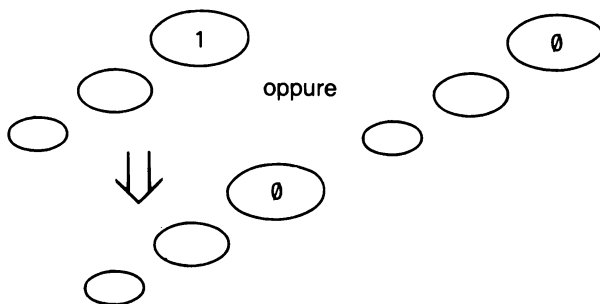
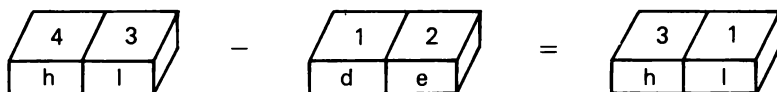
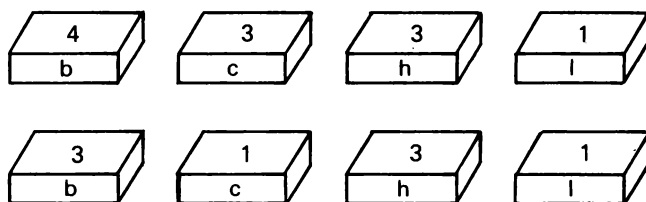
Lo stesso segnale di riporto è una bestia strana: non vi si può accedere direttamente e quindi non è possibile per esempio caricare in A il valore di questo segnale. D'altra parte esistono istruzioni che tengono conto anche del segnale di riporto. Lo stato del segnale di riporto può comunque essere controllato e quindi può essere "alzato" o "abbassato" a volontà. È proprio uno di quei comandi che considerano anche il segnale di riporto che noi useremo per la somma dei byte più significativi. Useremo un'istruzione che somma E ad H più il valore del segnale di riporto. Dato che il valore di questo segnale viene fissato durante l'addizione dei byte meno significativi, e che esso non viene modificato da un trasferimento di registri, otterremo in questo modo il risultato corretto.

Torniamo al nostro esercizio. Eravamo rimasti al punto in cui i due byte meno significativi erano stati sommati, ed eravamo pronti per sommare quelli più significativi. Poiché non possiamo sommare D con H, trasferiamo H in A.

Adesso sommiamo il registro D al registro A, tenendo conto dello stato del segnale di riporto.

L'istruzione successiva è quindi quella di "sommare D più il riporto al registro A".

Adesso abbiamo ottenuto il risultato di D più H più il riporto nel registro A, e vogliamo riportarlo nel registro H, cosicché il risultato finale sia contenuto in HL. Il trasferimento si fa così:

IL RISULTATO È ORA FINALMENTE CONTENUTO IN HL!!

L'addizione è ora completa, solo che la risposta deve essere resa accessibile dal BASIC. Possiamo mettere il valore di HL in BC cosicché esso possa essere stampato sul video o trasferito ad una variabile del BASIC, oppure possiamo conservare il risultato in locazioni di memoria, cui si può accedere dal BASIC. Tra queste due opzioni quella che prevede l'uso di BC potrebbe sembrare la più conveniente ai nostri scopi. Tuttavia, dato che useremo i registri BC durante una parte del programma, per la sottrazione, dovremo usare il metodo delle locazioni di memoria. Due locazioni di memoria sono necessarie per questa addizione a due byte (16 bit). Le locazioni in cui potremo caricare la risposta possono essere in qualsiasi punto della RAM, però bisogna prima assicurarsi di non scrivere sopra qualche altro programma. Se avete qualche dubbio andate a rivedervi la sezione con la mappa della memoria. Per i vostri scopi andranno bene le locazioni 27004 e 27005.

NOTA:
Il computer memorizza prima il byte meno significativo e poi quello più significativo, quindi ricordatevi di moltiplicare per 256 il contenuto del secondo indirizzo, non quello del primo.

Occupiamoci adesso della sottrazione di un registro a 16 bit da un altro, usando un comando di sottrazione per coppie di registri. Come? Consideriamo come punto di partenza che i due numeri richiesti siano stati caricati nelle coppie di registri DE e HL; l'obiettivo sarà allora quello di sottrarre il valore di DE dal valore di HL e di lasciare il risultato in HL. Per fare ciò basta un solo comando, ma ricordate che questo comando tiene conto anche del segnale di riporto. Se l'ultima istruzione eseguita ha fatto sì che questo segnale sia posto a 1, il nostro risultato potrebbe essere sbagliato, quindi dobbiamo assicurarci che lo stato del segnale sia 0. Ciò può essere fatto in modo indiretto. Per il momento annotiamoci soltanto l'istruzione "rimetti a 0 il segnale di riporto", cioè:

Con il segnale di riporto posto uguale a 0, non ci sarà influenza sul risultato della sottrazione. Possiamo procedere e sottrarre (con riporto) DE da HL, come indicato nel disegno che segue:

Siamo così riusciti a sottrarre il valore di DE da HL, lasciando il risultato in HL. Ora non ci resta che riportare il risultato ad una forma accessibile tramite il BASIC. Prima abbiamo evitato di usare la coppia di registri BC nel caso ci fosse necessaria per la costruzione del programma, un metodo forse complicato per far capire una cosa, ma tuttavia molto utile. Ora che è evidente che BC è libero per l'uso, possiamo servircene per il nostro scopo. Trasferisci H in B e L in C. Il risultato che prima era contenuto in HL è ora contenuto in BC. Vedi sotto:

Il modo in cui funziona il nostro programma è ora ben definito, ed il passo successivo è il trasferimento dei valori originari dal BASIC alla memoria, e poi dar controllare dalla routine in linguaggio macchina il trasferimento del contenuto di ciascuna locazione alla corretta coppia di registri.

Trasferimento dei valori iniziali dal Basic per l'utilizzo nell'ambito del linguaggio macchina

Il primo requisito è la conversione di quattro numeri da 16 bit in coppie di numeri ad 8 bit; questi potranno poi essere messi in memoria e poi richiamati dal linguaggio macchina. È abbastanza facile determinare questa procedura in esadecimale. Prendete le prime due cifre e trovate il loro equivalente in "base 256". Il risultato vi darà le due metà del numero a 16 bit. Alla fine di questo libro troverete una tabella di tutti i numeri binari, decimali ed esadecimali da 0 a 255. Non esitate a fare riferimento a queste tabelle: esse vi permetteranno di far minore fatica.

Ricordatevi poi che il computer stesso memorizza prima il byte meno significativo e poi quello più significativo. Fate anche voi lo stesso. Mentre da un lato ciò non altera il risultato, purché nel programma vengano fatte le opportune modifiche, dall'altro lato ciò vi aiuterà ad abituarvi ad operare in questo modo.

Gli otto numeri ad 8 bit possono essere memorizzati in qualsiasi punto della RAM, purché non interferiscano con altre cose. Ho già scelto di usare l'indirizzo 27000 come punto di partenza. Per l'addizione:

L = 27000
H = 27001
E = 27002
D = 27003

e per la sottrazione:

L = 27006
H = 27007
E = 27008
D = 27009

Il risultato dell'addizione verrà tenuto negli indirizzi 27004 e 27005.

Avendo convertito i numeri a 16 bit in due numeri a 8 bit, è molto facile mettere i numeri nei vari indirizzi in preparazione per l'esecuzione della routine in linguaggio macchina.

Come entriamo nell'esecuzione del codice in linguaggio macchina?

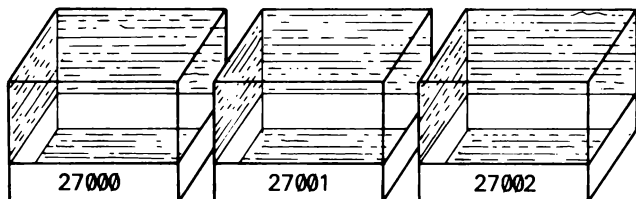
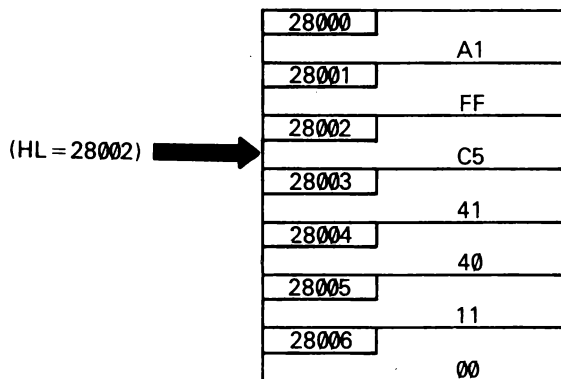
Non è possibile entrare nel linguaggio macchina mediante le istruzioni "GOTO" o "GOSUB", dato che entrambe servono solo per programmi in BASIC. Esiste invece in BASIC un comando molto particolare per l'utilizzo specifico da parte dell'utente di linguaggio macchina: "USR", che sta per "User Sub Routine". Nell'ambito del BASIC, "USR" non è un comando completo, ed è quindi necessario usare come prefisso uno dei seguenti comandi:

"PRINT", e così si può stampare sul video il valore di ritorno contenuto nei registri BC;

"LET", con cui si può assegnare ad una variabile il contenuto della coppia di registri al momento del ritorno;

"RANDOMIZE", che semplicemente prende come "seme" per il generatore di numeri casuali il risultato contenuto in BC.

Sia "PRINT" che "LET" sono dispendiosi di memoria, per cui il comando "RANDOMIZE" è quello generalmente più usato per entrare in una subroutine in linguaggio macchina. Una routine USR è molto simile ad una routine GOSUB, nel senso che si tratta di una subroutine; per uscirne si dovrà allora usare l'istruzione "RETURN", che per quanto riguarda il linguaggio macchina è stata abbrebiata in "RET".

Dobbiamo vedere adesso come si possa accedere alle costanti che sono state memorizzate per mezzo del BASIC. Il miglior modo per rappresentare i numeri memorizzati nelle loro locazioni è per mezzo di un disegno come il seguente:

A prima vista la mossa successiva sembrerebbe essere quella di usare un comando per caricare il contenuto dell'indirizzo 27000 per esempio nel registro C. Statene certi, ciò richiederebbe l'uso di quattro routine piuttosto lunghe, e non è quindi consigliabile. Ci serviamo invece della possibilità di caricare in un registro il contenuto di una locazione il cui indirizzo sia contenuto in HL. Questo metodo, nel quale si può aumentare o diminuire il valore di HL in modo da selezionare i byte da caricare nei registri, viene chiamato "metodo dell'utilizzo di HL come puntatore". Il funzionamento del puntatore viene chiarito nel disegno che segue:

Adesso mettiamo in pratica tutto ciò.

— Fissa il puntatore a 27000: "carica 27000 in HL".

— Carica la prima metà del primo numero a 16 bit in C: "trasferisci in C il contenuto della locazione HL".

— Sposta il puntatore in avanti di una locazione: "aumenta di 1 il valore di HL".

— Carica la seconda metà del primo numero a 16 bit in B: "trasferisci in B il contenuto della locazione HL".

— Sposta il puntatore in avanti di una locazione: "aumenta di 1 il valore di HL".

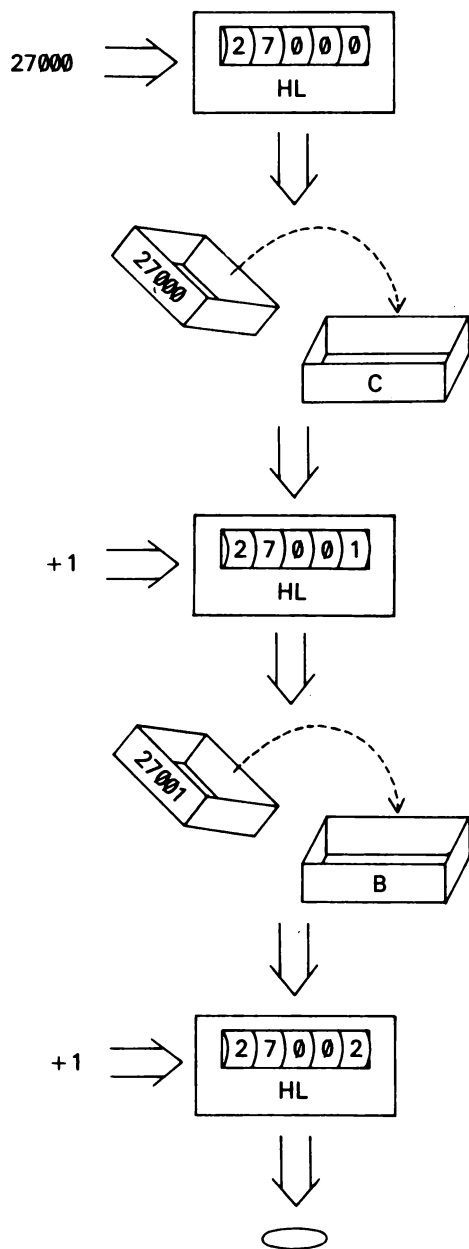
— Trasferisci in E la prima metà del secondo numero a 16 bit: "trasferisci in E il contenuto della locazione HL".

— Sposta il puntatore in avanti di una locazione: "aumenta di 1 il valore di HL".

— Trasferisci in D la seconda metà del secondo numero a 16 bit: "trasferisci in D il contenuto della locazione HL".

Il trasferimento è così completo; resta solo il problema che uno dei due numeri deve essere caricato in BC poiché HL veniva usato come puntatore, dato che essa è l'unica che può contenere l'indirizzo di una locazione cui bisogna accedere. Eseguite ora le seguenti due istruzioni: "trasferisci B ad H" e "trasferisci C ad L". Le costanti che erano state memorizzate dal BASIC si trovano ora nelle giuste coppie di registri per il linguaggio macchina.

Poi c'è il trasferimento delle costanti dal BASIC al linguaggio macchina per la parte della sottrazione. Notate i grandi vantaggi del sistema del puntatore. Una delle caratteristiche più interessanti di questo sistema è la facilità con cui si può far accedere il computer ad indirizzi diversi semplicemente cambiando il contenuto iniziale dei registri HL, che servono da puntatore. Un altro aspetto del sistema del puntatore è che si può accedere ad una locazione dopo l'altra non solo andando avanti, ma anche procedendo all'indietro (riducendo il valore del puntatore). Ciò viene ottenuto con l'istruzione "DEC". Questo procedimento sarà più chiaro con un piano particolareggiato sul caricamento dei valori iniziali dalla memoria ai registri.

27ØØØ
27ØØØ
HL
27ØØØ
C
+1
27ØØ1
HL
27ØØ1
B
+1
27ØØ2
HL

— Carica nel puntatore il valore iniziale: "carica 27009 in HL".

— Trasferisci in D la seconda metà del secondo numero a 16 bit: "trasferisci in D il contenuto della locazione HL".

— Sposta il puntatore indietro di una locazione: "diminuisci di 1 il valore di HL".

— Trasferisci in E la prima metà del secondo numero a 16 bit: "trasferisci in E il contenuto della locazione HL".

— Sposta il puntatore indietro di una locazione: "diminuisci di 1 il valore di HL".

— Trasferisci in B la seconda metà del primo numero a 16 bit: "trasferisci in B il contenuto della locazione HL".

— Sposta il puntatore indietro di una locazione: "diminuisci di 1 il valore di HL".

— Trasferisci in C la prima metà del primo numero a 16 bit: "trasferisci in C il contenuto della locazione HL".
Abbiamo ora lo stesso problema di prima: dobbiamo trasferire in HL il contenuto di BC, dato che BC veniva usato per tenere i valori temporaneamente, mentre HL veniva usato come puntatore. Quindi:

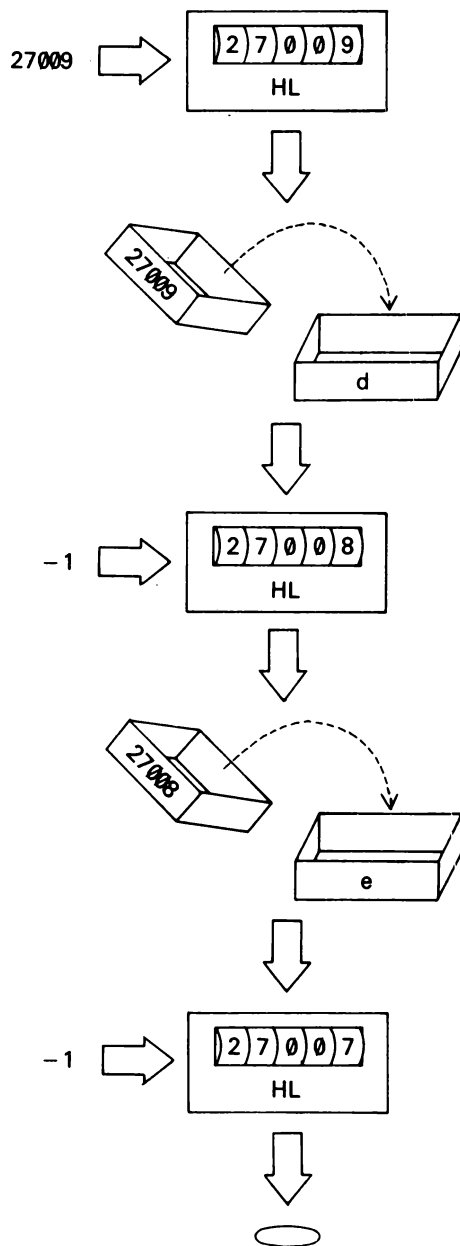
— "trasferisci B in HL".

— "trasferisci C in L".

Il disegno che segue mostra esattamente cosa sta succedendo.

Il procedimento per il passaggio dei dati dal BASIC al linguaggio macchina è così completo. Essi sono stati elaborati e rimemorizzati in modo che siano accessibili dal BASIC. Quindi l'effettiva scrittura del programma è quasi finita. Rimane solo il problema del trasferimento del controllo dal linguaggio macchina al BASIC.

Il comando "USR" che usiamo per passare il controllo dal BASIC al linguaggio macchina è già stato spiegato ed è stato paragonato al comando "GOSUB" del BASIC. Il puntatore di riga deve ora essere spsotato dalla riga corrente alla subroutine indicata. Quando la subroutine è terminata si usa il comando "RETURN" per dire al computer di andare alla riga che segue quella contenente l'istruzione "GOSUB" di chiamata della subroutine. Per esempio se tutte le righe sono numerate con passo di 10 e l'istruzione "GOSUB" è alla riga 10, l'istruzione "RETURN" farà sì che il programma continui l'esecuzione dalla riga 20. In

27009
2 7 0 0 9
HL
27009
d
−1
2 7 0 0 8
HL
27008
e
−1
2 7 0 0 7
HL

linguaggio macchina si usa la stessa istruzione "RETURN" per trasferire il controllo dall'istruzione attuale della routine in linguaggio macchina alla riga di BASIC successiva a quella contenente la chiamata con l'istruzione "USR". Con un'istruzione "USR" alla riga 10, e con la riga successiva identificata dal numero 20, quando la routine in linguaggio macchina è finita incontrando un'istruzione "RET" il programma BASIC continuerà ad eseguire dalla riga 20.

Dunque, l'ultima istruzione di un programma in linguaggio macchina sarà "ritorna al BASIC".

Mettiamo tutto insieme

È buona pratica che prima di procedere si prepari una lista chiara e logica di tutte le istruzioni che costituiscono il programma.

La prima parte della lista che segue mostra le istruzioni BASIC necessarie per il trasferimento dei valori originari; la seconda parte indica l'effettivo funzionamento del programma in linguaggio macchina, includendo il trasferimento in ingresso ed in uscita e le routine di addizione e sottrazione; la parte finale mostra come accedere alle risposte del BASIC.

Basic:

— Trasferisci il primo numero per l'addizione nelle locazioni 27000 e 27001 (LH)

— Trasferisci il secondo numero per l'addizione nelle locazioni 27002 e 27003 (ED)

— Trasferisci il primo numero per la sottrazione nelle locazioni 27006 e 27007 (LH)

— Trasferisci il secondo numero per la sottrazione nelle locazioni 27008 e 27009 (ED)

— Salta alla routine in linguaggio macchina.

Linguaggio macchina:

— Carica 27000 nella coppia di registri HL
— Trasferisci in C il contenuto della locazione HL
— Aumenta di 1 il valore di HL

— Trasferisci in B il contenuto della locazione HL
— Aumenta di 1 il valore di HL
— Trasferisci in B il contenuto della locazione HL
— Aumenta di 1 il valore di HL
— Trasferisci in E il contenuto della locazione HL
— Aumenta di 1 il valore di HL
— Trasferisci in D il contenuto della locazione HL
— Trasferisci B in H
— Trasferisci C in L
— Trasferisci L in A
— Somma E ad A
— Trasferisci A in L
— Trasferisci H in A
— Somma D più il riporto di A
— Trasferisci A in H
— Trasferisci HL nelle locazioni 27004 e 27005
— Nessuna operazione
— Nessuna operazione
— Carica 27009 nella coppia di registri HL
— Trasferisci in D il contenuto della locazione HL
— Diminuisci di 1 il valore di HL
— Trasferisci in E il contenuto della locazione HL
— Diminuisci di 1 il valore di HL
— Trasferisci in B il contenuto della locazione HL
— Diminuisci di 1 il valore di HL
— Trasferisci in C il contenuto della locazione HL
— Trasferisci B in H
— Trasferisci C in L
— Riporta a zero il segnale di riporto
— Sottrai con riporto la coppia DE dalla coppia HL
— Trasferisci H in B
— Trasferisci L in C
— Ritorna al BASIC.

BASIC:

— Stampa il valore finale della coppia BC (risultato della sottrazione).
— Prendi il risultato dell'addizione con il comando PEEK, e quindi stampalo.

Conversione in L

Adesso abbiamo chiarito per bene ciò che ci aspettiamo venga eseguito dal computer, e siamo arrivati al punto in cui dobbiamo convertire le nostre istruzioni in mnemoniche del linguaggio assembly, che potranno quindi essere immessi in

un assemblatore. È una buona idea a questo punto ricercare i codici esadecimali delle varie istruzioni, che potranno essere inseriti nel computer tramite il monitor in BASIC oppure con un programma debugger. Questa operazione può essere utile anche come controllo. Qui di seguito viene riportata una tabella completa con le varie istruzioni, le relative mnemoniche in assembler ed il loro codice esadecimale:

Istruzioni a parole	**Linguaggio Assembler**	**Codici esadecimali**
— Carica 27000 in HL	LD HL, 27000	21 78 69
— Contenuto della locazione HL in C	LD C, (HL)	4E
— Aumenta di 1 il valore di HL	INC HL	23
— Contenuto della locazione HL in B	LD B, (HL)	46
— Aumenta di 1 il valore di HL	INC HL	23
— Contenuto della locazione HL in E	LD E, (HL)	5E
— Aumenta di 1 il valore di HL	INC HL	23
— Contenuto della locazione HL in D	LD D, (HL)	56
— Trasferisci B in H	LD H,B	60
— Trasferisci C in L	LD L,C	69
— Trasferisci L in A	LD A,L	7D
— Somma E ad A	ADD A,E	83
— Trasferisci A in L	LD L,A	6F
— Trasferisci H in A	LD A,H	7F
— Somma D più il riporto di A	ADC A,D	8A
— Trasferisci A in H	LD H,A	67
— Contenuto di HL nelle locazioni 27004 e 27005	LD (27004),HL	22 7C 69
— Nessuna operazione	NOP	00
— Nessuna operazione	NOP	00
— Carica 27009 in HL	LD HL,27009	21 81 69
— Contenuto della locazione HL in D	LD D, (HL)	56
— Diminuisci di 1 il valore di HL	DEC HL	2B
— Contenuto della locazione HL in E	LD E, (HL)	5E
— Diminuisci di 1 il valore di HL	DEC HL	2B
— Contenuto della locazione HL in B	LD B, (HL)	46
— Diminuisci di 1 il valore di HL	DEC HL	2B
— Contenuto della locazione HL in C	LD C, (HL)	4E
— Trasferisci B in H	LD H,B	60
— Trasferisci C in L	LD L,C	69
— Riporta a zero il segnale di riporto	AND 0	E6 00
— Sottrai con riporto la coppia DE dalla coppia HL	SBC HL,DE	ED 52
— Trasferisci H in B	LD B,H	44
— Trasferisci L in C	LD C,L	4D
— Ritorna al BASIC.	RET	C9

Il "dry-run"

È arrivato il momento di controllare attentamente il programma, per trovare gli errori che possono esservi entrati a nostra insaputa. Nel caso di errori gravi è possibile che il computer "salti" senza darvi alcuna informazione per determinare cosa non ha funzionato. Dopo che si è faticato tanto per introdurre il programma nel computer non è certo divertente aver a che fare con un errore. Questo è dunque il momento di compiere il "dry-run".

L'esecuzione in "dry-run" di un programma non è altro che l'esecuzione di un programma su carta. Potete servirvi, per esempio, di una tabella con tutti i registri, inserendovi i valori contenuti in ciascuno di essi dopo ogni istruzione, annotandovi anche lo stato del segnale di riporto. Di solito con questo metodo gli errori saltano fuori abbastanza rapidamente. Un'altra qualità di questo sistema di controllo è che esso consente di preparare un documento che mostra come effettivamente funziona il programma, facilitando così un suo successivo controllo o modifica. Qui di seguito viene riportata una semplice tabella che vi potrà essere di aiuto durante il "dry-run" dei vostri programmi.
Prendete in considerazione ciascuna mnemonica dell'assembler, decidete cosa fa e scrivete nella tabella 1 il risultato dell'istruzione. Per esempio, se si incontra il comando "LD A,40" scrivete 40 nella colonna A. Usando la tabella in questo modo, si otterrà un'immediata verifica del contenuto dei registri. Allargare la tabella per contenere, oltre ai registri, tutte le locazioni di memoria utilizzate dal programma può essere utile per acquistare familiarità con il suo uso.

H	L	D	E	B	C	A	Riporto (se noto)

Immissione nell'assembler

Sebbene in questo caso sia stata data la lista dei codici esadecimali, non è una pratica usuale tradurre un programma nei suoi codici, se si possiede un assembler, poiché ciò vorrebbe dire fare a mano il lavoro dell'assembler. A questo punto abbiamo sia l'assembler che un programma da assemblare, e possiamo perciò procedere.

Dopo aver caricato l'assembler con il comando LOAD, gli si devono fornire le istruzioni per il funzionamento. In questo caso si dà un comando GO per indicare che c'è un programma in attesa di essere assemblato. Quindi si usa l'istruzione ORG insieme all'indirizzo di una locazione di memoria per identificare la posizione in cui dovrà essere conservato il programma assemblato. Anche se ora si è pronti per inserire le mnemoniche dell'assembler (precedute dalla parola REM), è sempre bene aggiungere un commento prima dell'inizio del programma; questo potrà servire in seguito ad identificare il programma stesso. Questa possibilità si ottiene mettendo un punto esclamativo dopo REM, e facendo seguire il commento. Per il nostro programma può essere appropriato scrivere "routine di addizione e sottrazione".

Ciascuna mnemonica deve essere scritta attentamente assicurandosi di non commettere errori. È possibile mettere più istruzioni in ogni riga, separandole con un punto e virgola, ma è molto più facile leggere la routine quando ogni riga contiene una sola istruzione. Ecco come dovrebbe apparire la lista delle istruzioni una volta immesse nell'assembler:

VERSIONE FINALE

```
ld hl,27000          217869
ld c,(hl)                4E
inc hl                   23
ld b,(hl)                46
inc hl                   23
ld e,(hl)                5E
inc hl                   23
ld d,(hl)                56
ld h,b                   60
ld l,c                   69
ld a,l                   7D
add a,e                  83
ld l,a                   6F
ld a,h                   7C
adc a,d                  8A
ld h,a                   67
ld (27004),hl        227C69
```

```
nop              00
nop              00
ld hl,27009      218169
ld d,(hl)        56
dec hl           2B
ld e,(hl)        5E
dec hl           2B
ld b,(hl)        46
dec hl           2B
ld c,(hl)        4E
ld h,b           60
ld l,c           69
and 0            E600
sbc hl de        ED52
ld b,h           44
ld c,l           4D
ret              C9
```

Siamo ora quasi pronti per assemblare la routine; prima, comunque, ci sono ancora alcune cose da fare.

Aggiungete innanzitutto un commento nella penultima riga, per indicare che questa è la fine della routine. Le istruzioni GO, FINISH e ORG sono molto importanti per l'assembler, che si rifiuterà di funzionare se queste mancano. L'assemblaggio si interromperà perciò con un messaggio di errore. Se, inoltre, gli spazi sono troppi o troppo pochi, l'assembler potrà incorrere in errori di interpretazione, problema questo piuttosto grave perché difficile da individuare.

Una volta che il programma sia stato assemblato è bene farne una copia su nastro o microdrive, in modo da non perdere, in caso di incidenti, tutto il lavoro fatto. Per conservare il programma assemblato su cassetta, scrivete:

SAVE "nome" CODE xxxxx,yyyyy

dove xxxxx sta ad indicare la locazione di memoria da cui inizia la routine, e yyyyy il numero di byte della routine stessa. La parola CODE è la parte essenziale, poiché dice al computer che ciò che deve copiare è scritto in linguaggio macchina. Per verificare ciò che si è copiato si scrive: VERIFY " " CODE. Per fare una copia del programma BASIC, che contiene le mnemoniche e che è molto semplice da modificare e ricompilare, si procede esattamente come al solito con un normale programma, cioè: SAVE "nome".

Per caricare poi nel computer un programma in linguaggio macchina nella sua forma esadecimale si deve scrivere: LOAD "nome" CODE. In questo modo si carica un programma in linguaggio macchina chiamato "nome", il quale verrà

posto in memoria allo stesso indirizzo da cui era stato copiato su nastro e dischetto. Per ricaricare un normale programma BASIC, cioè il programma contenente le mnemoniche, si scrive: LOAS "nome".

Il programma BASIC usato per tenere le mnemoniche prima dell'assemblaggio ha così esaurito la sua utilità e può essere cancellato. Non cercate di farlo però tramite il comando "NEW", altrimenti perdereste tutto. Usate invece il comando "CLEAR xxxxx", dove xxxxx è la locazione a partire dalla quale si vuole proteggere la memoria, meno 1: in questo modo il RAM TOP viene riposizionato al punto desiderato e il programma BASIC può così essere cancellato con "NEW". State pur certi che in questo modo il vostro programma in linguaggio macchina è al sicuro, così come l'assembler, che si trova ancora in memoria.

Vediamo adesso il programma di controllo che organizza il trasferimento delle variabili alle locazioni di memoria, la lettura dei risultati e la loro stampa sul video.

```
10  POKE 27000,4:POKE 27001,1:
    POKE 27002,3:POKE 27003,2
20  POKE 27006,7:POKE 27007,1:
    POKE 27008,8:POKE 27009,9
30  PRINT "SUB =";USR 28000
40  PRINT "ADDIZIONE=";PEEK 27004+
    (PEEK 27005)*256
50  STOP
```

È molto conveniente fare in modo che questo programma di controllo in BASIC venga caricato in memoria quasi contemporaneamente al programma in linguaggio macchina. Ciò viene ottenuto facendo iniziare da sé l'esecuzione del programma BASIC, con la sua prima riga contenente l'istruzione per il caricamento del programma in linguaggio macchina. Così essi potranno essere conservati uno vicino all'altro nella cassetta.

Dopo aver verificato la copia del programma, resta solo da farlo eseguire. Per far ciò basta un'istruzione "GOTO" o "RUN" al programma BASIC alla giusta riga, dato che poi l'istruzione "PRINT USR 28000" è inclusa all'interno del programma BASIC. Allora potrete fare qualche prova con diversi valori nei registri e osservare (prevedere?) i risultati.

CAPITOLO 8

I SALTI

Lo sviluppo di una buona tecnica è stato l'interesse essenziale di quanto si è detto finora. È giunto ora il momento di estendere il repertorio dei comandi disponibili per l'uso dei nostri programmi: certamente essi sono necessari per dare libero sfogo ai vostri istinti creativi. Più cose si conoscono, più se ne possono usare. Sarà via via sempre più evidente che anche solo con pochi comandi si possono ottenere numerose combinazioni di programmi diversi, ottenendo così una grande versatilità di programmazione. Questo capitolo sarà dedicato a salti semplici, salti condizionati, salti relativi, salti relativi condizionati, chiamate e chiamate condizionate.
Inoltre, si prenderà in considerazione il registro PC, l'istruzione RET e altri aspetti connessi con il funzionamento interno del computer.

Il registro PC

L'organizzazione dei registri della CPU è già stata spiegata in termini generali, ed una particolare attenzione è stata data a quelli che possono più facilmente essere manipolati (HL, BC, etc.). Esistono altre coppie di registri cui non si può accedere direttamente, oppure ai quali si può accedere solo attraverso operazioni piuttosto elementari. Ciò deriva dall'esigenza della CPU di tenere riservato uno spazio dove conservare le proprie variabili in modo che siano accessibili ma allo stesso tempo protette da interferenze esterne. Il registro PC è un esempio di questo genere di registri. Esso contiene l'indirizzo della locazione di memoria in cui si trova il comando che attualmente viene eseguito, e funziona quindi da puntatore per la CPU fornendole la locazione d'inizio dell'istruzione successiva.

Quando si accende il computer all'inizio di una sessione di lavoro, il valore del registro PC è 0000, cosicché il primo comando che il computer esegue immediatamente è quello contenuto nella prima locazione di memoria.
Ecco perché la ROM del Sinclair è assegnata ai primi 16K di memoria. In altri termini, tramite l'utilizzo di un chip di tipo ROM, i valori contenuti nelle locazioni da 0 a 16383 sono predeterminati, e per questo motivo ogni volta che si accende

il computer esso inizia subito ad eseguire quella che è nota come "routine di inizializzazione". Quando poi ci si trova in una routine di linguaggio macchina il registro PC contiene sempre l'indirizzo della locazione dell'istruzione corrente, per cui quando essa è terminata PC cambia il suo valore secondo la lunghezza dell'istruzione stessa. Si può conoscere così la locazione dell'istruzione successiva. Si può assumere allora che, modificando dall'esterno il valore del registro PC, è possibile determinare a priori la locazione da cui la CPU prenderà la sua istruzione successiva.

Il primo problema da superare è che non esistono comandi che operano direttamente sul contenuto del registro PC, come esistono invece per esempio per i registri HL e BC. Per nostra fortuna c'è invece un'istruzione che modifica il valore all'interno del registro PC, e cioè l'istruzione JUMP., abbreviata con JP. Essa funziona in modo molto simile all'istruzione GO TO del BASIC, semplicemente caricando nel registro PC il valore desiderato, cioè l'indirizzo da cui si prenderà l'istruzione successiva. In BASIC, se ci si trova alla linea 100 e si vuole che l'istruzione successiva venga letta alla linea 1050, si userà alla linea 100 l'istruzione GO TO 1050. In linguaggio macchina l'operazione è esattamente la stessa. Dunque, se abbiamo una breve routine dalla locazione 32000, la procedura da seguire sarà questa:

30000 – 30012	: prima routine
30013 : JP 32000	: salta alla locazione 32000
32000 –	: seconda routine

Ecco un esempio dell'uso di questa istruzione:

```
org 30000
30000 3E 00       ld a,0
30002 06 07       ld b,7
30004 C3 00 7D    jp 32000
30007 60          ld h,b
30008 6F          ld l,a
30009 C9          ret
```

Sebbene sia senz'altro molto utile poter saltare da una parte di memoria ad un'altra, ed eseguire poi le istruzioni da quella parte di memoria, in quanti casi è effettivamente necessario scrivere una routine in due parti diverse di memoria? Non sarebbe molto più utile essere in grado di far eseguire un comando da un'altra parte della memoria solo se una certa situazione è vera o falsa? Per esempio in un programma BASIC con un calcolo lungo e complicato, si richiede al computer di indicare se il risultato sia maggiore o minore del previsto: si potrà usare una routine come la seguente:

```
500     IF risposta maggiore del previsto THEN GOTO 1000
510     IF risposta minore o uguale al previsto THEN GOTO
        1500
1000    PRINT "maggiore di"
1010    STOP
1500    PRINT "minore o uguale a"
1510    STOP
```

Si noti che in questo programma BASIC l'istruzione "STOP" è stata usata per completare la routine, mentre non esiste in linguaggio macchina un'istruzione equivalente, se non altro perché non si richiede mai un arresto totale: alla fine di una routine in linguaggio macchina si dovrà sempre fare ritorno al BASIC. Nel suo uso normale, l'istruzione "RET" restituisce al BASIC il controllo del computer.

Eccovi ora una lista delle sei istruzioni di salto condizionato più frequentemente usate:

JP Z	: salta se uguale a zero;
JP NZ	: salta se diverso da zero;
JP C	: salta se c'è riporto;
JP NC	: salta se non c'è riporto;
JP M	: salta se negativo;
JP P	: salta se positivo.

"JP Z" significa che l'istruzione successiva sarà presa dalla locazione indicata soltanto quando viene "alzato" (posto pari a 1) il segnale di azzeramento, ossia quando il risultato dell'ultimo calcolo eseguito era zero. Se il risultato non è nullo, e cioè questa condizione è falsa, verrà presa l'istruzione della successiva locazione di memoria.

Per esempio:

```
org 28000
28000 3E 00          ld a,0
28002 C6 00          add a,0
28004 CA 6B 6D       jp z,28011
28007 3D             dec a
28008 C3 62 6D       jp 28002
28011 C9             ret
```

In questo programma il test serve per determinare se sommando zero all'accumulatore si ottiene un risultato nullo. Se sì, il controllo del programma torna al BASIC;

altrimenti il valore dell'accumulatore viene diminuito di 1 e lo stesso test viene riprovato. Si continua così finché l'accumulatore sia pari a zero. Assicuratevi sempre che l'istruzione usata prima di un salto condizionato vada effettivamente ad incidere sullo stato di quel particolare segnale. Se avete dei dubbi consultate la tabella alla fine del libro.

"JP NZ" significa che l'istruzione successiva sarà presa dalla locazione indicata soltanto quando viene abbassato (posto pari a 0) il segnale di azzeramento, ossia quando il risultato dell'ultimo calcolo eseguito non era zero. Se, come nell'esempio di prima, la precedente istruzione era: "aggiungi zero", seguita da "JP NC", 32000" la routine sarebbe così:

```
29999 3D           dec a
30000 FD 69        add 0
30002 CA "end"     jp nz,29999
30005 C3 "32000"   jp 32000
```

L'equivalente istruzione in BASIC sarebbe:

IF (A più 0 è uguale a 0) THEN GOTO 32000.

"JP C" significa "salta solo se viene alzato il segnale di riporto. In altri termini, se l'ultimo calcolo ha provocato un'uscita fuori scala, il segnale di riporto viene alzaro, ovvero posto pari a 1, e la condizione per il salto risulta verificata. Un semplice esempio può essere dato dall'istruzione di saltare ad un nuovo indirizzo qualora la somma di A più B sia maggiore di 255, altrimenti di continuare all'indirizzo seguente. Ecco un programma con questa istruzione:

```
org 30000
30000 80              add a,b
30001 DA 30 75        jp c,30000
```

L'equivalente in BASIC sarebbe: IF A più B è maggiore di 255 THEN GOTO 30000.

In questo modo si ottiene un semplice controllo della possibilità di errori derivanti dal fatto che la somma di due numeri è troppo elevata per essere tenuta nei registri della CPU.

L'istruzione "JP NC" è molto simile, tranne che si salta solo se il segnale di riporto viene abbassato. Si sottopone a test il fatto che il risultato dell'ultimo

calcolo sia andato fuori scala; se ciò non è avvenuto, se cioè la somma di due numeri è minore di 256, si salta all'indirizzo indicato. Ecco un esempio:

```
org 30000
30000 80            add a,b
30001 D2 30 75      jp nc,30000
```

L'equivalente in BASIC sarebbe: IF A + B<256 THEN GOTO 30000.

Le ultime due condizioni che sono disponibili per le istruzioni di salto semplice sono la positività o negatività. Esse riguardano lo stato del segnale di segno, che può indicare segno positivo o negativo. Poiché dobbiamo ancora parlare di questo segnale, ci limitiamo a dire che "JP P" significa "salta se il segno indica un numero positivo"; "JP M" significa invece "salta se il segno indica un numero negativo".

In tutti questi esempi sono stati usati indirizzi effettivi per mostrare esattamente cosa sta succedendo. Tuttavia, se si usa un assembler, oppure se si scrive un programma che potrà essere messo in qualunque punto della memoria, risulta molto più facile usare le "etichette". Per eseguire una routine una volta, e poi tornare di nuovo all'inizio del programma, si può mettere all'inizio l'etichetta "START", e alla fine della routine si fa eseguire l'istruzione "JP START". Usando un assembler non c'è bisogno di mettere gli effettivi indirizzi, dato che questo lavoro lo fa l'assembler, che predispone gli indirizzi per ogni parte del programma. È anche possibile contrassegnare con una etichetta certi particolari byte all'interno del programma, i quali potranno essere usati per conservare le risposte ai vari calcoli che vengono eseguiti. Questa funzione dell'assembler è una delle più utili, oltre a quella di tradurre le mnemoniche stesse. Qui di seguito viene riportato un breve esempio di come si possano usare le etichette nell'ambito di un programma. Non preoccupatevi se il modo di funzionamento di questo programma risulta un po' oscuro: considerate soltanto le etichette e il modo in cui esse vengono usate e definite.

1. Listato base per l'assembler ACS.

```
   1 REM go
   2 REM org 30000
  10 REM ld hl,22527
  20 REM ld b,192
  30 REM !CAMBIARE B&HL PER ALTR
I BLOCCHI DI SCHERMO"
  40 REM A;ld c,32
  50 REM B;ld e,(hl)
```

```
  60 REM rl (hl)
  70 REM dec hl;dec c
  80 REM jr nz,B
  90 REM bit 7,e
 100 REM jr z,C
 110 REM push hl; ld de,32
 120 REM add hl,de
 130 REM set 0,(hl)
 140 REM !RES 0,(HL) PER NON RIT
ORNO
 150 REM pop hl
 160 REM C;and a
 170 REM djnz A
 180 REM ret
 190 REM finish
```

2. Listato assemblato

```
org 30000
30000 21 FF 57     ld hl,22527
30003 06 C0        ld b,192
'CAMBIARE B&HL PER ALTRI
BLOCCHI DI SCHERMO
A
30005 0E 20        ld c,32
B
30007 5E           ld e,(hl)
30008 CB 16        rl (hl)
30010 2B           dec hl
30011 0D           dec c
30012 20 F9        jr nz,B
30014 CB 7B        bit 7,e
30016 28 08        jr z,C
30018 E5           push hl
30019 11 20 00     ld de,32
30022 19           add hl,de
30023 CB C6        set 0,(hl)
'RES 0,(HL) PER NON RITORNO
30025 E1           pop hl
C
30026 A7           and a
30027 10 E8        djnz A
30029 C9           ret
```

Ovviamente, è anche possibile saltare ad una locazione più indietro nel programma. Se per esempio ci si trova alla locazione 30000 e si vuole eseguire una routine alla locazione 28000, l'istruzione "JP 28000" posta alla 30000 è del tutto legittima. C'è soltanto una cosa a cui si deve prestare attenzione quando si usano questo tipo di salti all'indietro che creano un anello con se stessi: occorre stare attenti cioè a non usare salti incondizionati che portano a non incontrare mai un'istruzione di "RETurn". Questo è il modo più rapido per far sì che il computer entri in "anello infinito", senza speranza di uscirne! Per interrompere un tale

anello e tornare al BASIC bisognerebbe premere il tasto "Break", ma ciò non è possibile nel linguaggio macchina. Alcuni computer hanno la possibilità di reinizializzarsi, in modo da poter ricominciare senza perdere tutte le routine che hanno in memoria, ma ciò non è previsto per lo ZX 80, lo ZX 81 e lo ZX Spectrum. L'unico modo di uscire da un anello infinito nel linguaggio macchina con questi computer è quello di staccare la spina! Infatti, durante l'esecuzione del codice di linguaggio macchina lo Spectrum non registra la pressione del tasto "Break", e del resto di qualsiasi altro tasto, a meno che ovviamente non si sia scritto nel programma una routine di scansione della tastiera.

Abbiamo così esaminato i salti diretti che ci consentono di caricare un nuovo indirizzo nel registro PC, ovvero "Program Counter", in modo da modificare dall'esterno la locazione da cui verrà presa l'istruzione successiva. Esiste un altro tipo di istruzione di salto, utile per quando si vogliono scrivere brevi routine che non dipendono dall'area della memoria in cui esse vengono effettivamente caricate.

Questo nuovo tipo di salto viene chiamato "salto relativo". Il motivo di questo appellativo è che con questa istruzione è possibile passare a una locazione "più avanti o più indietro di un certo numero di locazioni rispetto a quella corrente". Invece di saltare per esempio dalla locazione 28005 alla 28000, si può usare un'istruzione di salto relativo di meno 5. Questo può sembrare un modo piuttosto complicato per passare da una locazione all'latra, ma ha l'effettivo vantaggio di permettere di posizionare la routine in qualsiasi punto della memoria.
Usando i salti relativi, dato che non si specifica effettivo per il salto, la routine può essere spostata dalla locazione 28000 alla 24000 o a qualsiasi altra locazione nella memoria senza bisogno di alcuna conversione.

Questa possibilità è particolarmente utile quando la routine può essere usata più di una volta nell'ambito di un programma più grande, oppure quando si deve convertire un programma per usarlo su un computer più potente. Analogamente, essa può essere utile quando non c'è abbastanza spazio nella memoria per una routine, ed è quindi necessario muoverla più in basso o più in alto per fare spazio alle altre cose. Il comando di salto relativo può in questi casi far risparmiare una notevole mole di lavoro. Esiste tuttavia una grossa limitazione a questo comando: è possibile saltare soltanto ad una locazione, che rispetto a quella corrente, si trovi ad un massimo di 129 byte in avanti oppure 126 all'indietro. Ciò può sembrare un po' troppo limitativo, ma in pratica "più 129 byte e meno 126 byte" fornisce di solito uno spazio di manovra veramente molto ampio nell'ambito di una normale routine. Inoltre può essere complicato calcolare spostamenti ("salti") più grandi di così, e si può facilmente sbagliare. Diciamo dunque che l'istruzione di salto relativo ("jump relative", abbreviate in "JR") viene considerata soprattutto come un'istruzione di tipo locale.

Il calcolo dello spostamento per un salto relativo è una cosa abbastanza semplice,

ma richiede una certa cura. L'uso di numeri negativi nell'ambito di un programma in linguaggio macchina non è possibile in modo diretto. Per un salto relativo con spostamento positivo (in avanti), cioè fra 0 e 129, il numero effettivo che deve essere fornito al computer è proprio quello dello spostamento; quando lo spostamento è negativo (all'indietro), cioè fra 0 e –126, il numero che deve essere inserito nel computer è pari a 256 più lo spostamento negativo. Usando l'assembler adottato per questo libro, invece, basta scrivere lo spostamento, sia esso positivo o negativo, e così non sarà necessario preoccuparsi di calcolare l'effettivo numero da fornire al computer nell'istruzione di salto in linguaggio macchina.

Quando viene eseguito il salto relativamente ad una istruzione, l'ordine che viene dato al computer è quello di passare ad un'istruzione posta un certo numero di byte più avanti di quella che normalmente verrebbe eseguita, cioè la successiva. In altre parole, se il computer eseguisse l'istruzione "JR 0" (salto relativo di 0 byte), il risultato sarebbe nullo, poiché il computer andrebbe in ogni caso ad eseguire il comando dell'indirizzo successivo. Questa istruzione vorrebbe dire semplicemente aumentare di 0 il valore del registro PC. Invece, poiché l'istruzione di salto relativo è lunga 2 byte, eseguendo l'istruzione "JR –2" si torna in effetti ad eseguire di nuovo questa stessa istruzione, entrando così in un anello infinito. L'unico modo per uscirne sarebbe quello di staccare la spina.

Qui di seguito viene riportato un esempio sull'uso del comando di salto relativo e anche di quanto esso sia utile se combinato con l'utilizzo di etichette nell'ambito dell'assembler.

```
org 30000
START
30000 3E 29         ld a,41
30002 06 28         ld b,40
30004 18 06         jr PARTE2
30006 00            nop
30007 00            nop
30008 00            nop
30009 00            nop
30010 00            nop
30011 00            nop
PARTE2
30012 0E 0E         ld c,14
30014 81            add a,c
30015 18 02         jr +2
30017 18 ED         jr START
30019 C9            ret
```

È interessante notare che in questo programma non si usano mai registri effettivi; la routine potrà quindi essere posta in qualsiasi punto della RAM del computer senza bisogno di alcuna modifica.

Esattamente come i salti condizionati, è possibile usare i salti relativi condizionati.

L'unica differenza è che i tipi di salti relativi condizionati a disposizione sono in numero inferiore, sono possibili soltanto i seguenti:

JR R : salto relativo se il segnale di azzeramento viene alzato
JR NZ : salto relativo se il segnale di azzeramento viene abbassato
JR C : salto relativo se il segnale di riporto viene alzato
JR NC : salto relativo se il segnale di riporto viene abbasato.

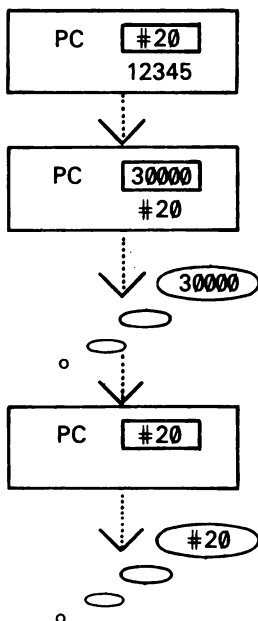
Queste istruzioni possono essere usate esattamente allo stesso modo delle analoghe versioni delle istruzioni di salto semplice, basta solo ricordarsi di usare un valore per uno spostamento invece che un effettivo indirizzo.
Nel BASIC, oltre all'istruzione GOTO c'è anche l'istruzione GOSUB: questa ci consente di far eseguire una subroutine invece che il programma principale e poi di ritornare al programma senza bisogno di un'altra istruzione GOTO. L'utilizzo di brevi subroutine è quindi molto semplice. Un altro vantaggio è la possibilità di riutilizzare una certa routine più volte nell'ambito di un programma. Per esempio, per eseguire una certa operazione, che richiede una serie di comandi, tre o quattro volte nello stesso programma, non è necessario scrivere più volte gli stessi comandi nel programma principale; si può invece far eseguire quell'operazione da una subroutine. La subroutine può trovarsi prima o dopo il programma principale, e vi si può accedere con il comando GOSUB del BASIC. Alla fine dell'esecuzione, l'istruzione RETURN alla fine della subroutine fa sì che il controllo del computer ritorni alla linea immediatamente dopo la frase GOSUB nel programma principale.
Anche nei programmi in linguaggio macchina è possibile usare delle subroutine, che funzionano esattamente come in BASIC. Per chiamare una subroutine in linguaggio macchina da un programma in linguaggio macchina bisogna usare l'istruzione "CALL", seguito dall'indirizzo in cui inizia la subroutine, oppure dalla etichetta con cui essa è contrassegnata, se si sta usando un assembler.

Ecco un esempio di come si può usare questa possibilità in un programma principale:

```
10 RANDOMIZE USR 30000

org 30000
30000 CD 6E 6D      call 28014
30003 C9            ret

org 28014
28014 3E 00         ld a,0
28016 06 00         ld b,0
28018 C9            ret

20 PRINT "ROUTINE COMPLETA"
```

Un altro modo di utilizzo dell'istruzione CALL è quello di eseguire subroutine già presenti nel chip di memoria ROM. Sebbene molte delle routine della ROM siano ben inserite nell'esecuzione del BASIC, e sebbene sia meglio lasciare al gc l'esecuzione di alcune altre routine, ve ne sono comunque molte che possono essere utili. Alcune di esse sono brevi, e quindi l'unico motivo per usare l'istruzione CALL è quello di risparmiare il tempo necessario a copiarle nei vostri programmi; molte invece sono lunghe e complesse, il che rende dispendioso, sia in termini di tempo che di spazio nella memoria, e ricopiarle. Inoltre l'uso dell'istruzione CALL elimina ogni possibilità di errore.

Dopo che una subroutine è stata eseguita, è necessario ritornare al programma principale. In BASIC ciò viene fatto con l'istruzione RETURN, in linguaggio macchina con RET. Ciò può inizialmente creare qualche confusione, dato che finora l'istruzione RET è stata usata per restituire il controllo del computer al BASIC alla fine di un programma in linguaggio macchina. Il fatto è che l'istruzionz RET serve semplicemente a restituire il controllo al punto in cui era stata usata una routine in linguaggio macchina mediante l'istruzione USR del BASIC, e poi si incontra l'istruzione RET, il controllo ritorna al BASIC. Viceversa, se si era eseguita l'istruzione CALL del linguaggio macchina, con RET si potrà restituire il controllo al programma principale in linguaggio macchina.

Quando viene usata la CALL, la locazione in cui essa è contenuta viene memorizzata nel computer, di modo che quando poi viene incontrata l'istruzione RET il

computer può ritornare al punto esatto nella memoria. Quando viene incontrata la RET e si restituisce il controllo al programma principale, il valore della locazione memorizzato nel computer viene sostituito dal valore precedente. In altri termini, quando si passa ad un programma in linguaggio macchina tramite l'istruzione USR del BASIC, il computer tiene memorizzato l'indirizzo della successiva istruzione del programma BASIC. Quando poi si passa dal programma in linguaggio macchina ad una subroutine in linguaggio macchina, mediante l'istruzione CALL, anche l'indirizzo della CALL viene memorizzato dalla CPU. Alla fine di questa subroutine si incontra l'istruzione RET ed il controllo torna all'indirizzo memorizzato dal computer: in questo caso si torna al programma principale in linguaggio macchina. Poiché ora la subroutine è stata completata, il valore tenuto memorizzato è quello che veniva tenuto prima della chiamata alla subroutine. Il disegno mostra più chiaramente cosa stia succedendo.

È possibile anche usare istruzioni di chiamata condizionata, proprio come i salti semplici e quelli relativi. Esse in effetti funzionano esattamente allo stesso modo delle istruzioni già spiegate, e non intendo approfondire oltre questo punto. Di nuovo, con le istruzioni CALL l'uso di etichette aumenta notevolmente la facilità con cui si possono usare le subroutine in un programma, ed aiuta la creazione di programmi ben strutturati. Diventa così possibile dividere l'intero programma in una serie di subroutine, la cui esecuzione viene controllata da un breve programma principale. Lavorando in questo modo si ottiene alla fine una lista del programma molto più comprensibile che altrimenti, e si può anche con facilità modificare ed estendere il programma stesso.
Purtroppo non esistono istruzioni del tipo di chiamate relative, e si devono quindi usare gli effettivi indirizzi, di modo che il programma non può essere spostato da un indirizzo ad un altro. Comunque, lo spostamento di un programma non è particolarmente difficile, poiché, come ricorderete, non è necessario che ci ricordiamo l'indirizzo cui fare ritorno dopo che la routine è conclusa. La subroutine può essere posta in un certo punto della memoria, e si sposta soltanto il programma principale. Le mnemoniche ed i codici esadecimali e decimali per le istruzioni di chiamata sono listati nell'Appendice B.

CAPITOLO 9

L'USO DELLO SCHERMO E DELLA TASTIERA

Questo capitolo sarà dedicato principalmente allo studio di come il computer e il suo utente possono interagire o "comunicare". Inizieremo esaminando brevemente l'organizzazione della tastiera ed il modo in cui si possono usare le variabili di sistema per avere facile lettura della tastiera. Vi saranno dati alcuni esempi pratici e qualche routine che potrete includere nei vostri programmi. Vedremo poi il modo in cui è organizzato il video e come le relative informazioni siano immagazzinate nella memoria. Poiché ciò richiede una grande porzione della memoria ad accesso casuale (RAM) del computer, riusciremo facilmente a modificare il contenuto del video in un modo piuttosto strano. Si cercherà quindi con attenzione il metodo per superare le difficoltà insite nello Spectrum, per definire nuovi caratteri e per poi scriverli sul video. Infine, verrà presentata una routine in linguaggio macchina che permette di fare il plot di un qualsiasi punto sullo schermo: data la sua sorprendente velocità, questa routine si dimostrerà molto più utile di quanto possa sembrare a prima vista.

La tastiera è in pratica divisa in righe e colonne, cosa che permette al computer di identificare con un minimo di istruzioni quale tasto venga premuto; inoltre, gli consente di registrare la pressione contemporanea di due tasti, purché si trovino su righe diverse. Esempi di questo caso sono il tasto "Caps Shift" (maiuscole) ed il tasto "Symbol Shift" (per i simboli grafici). Se essi si trovassero uno vicino all'altro e nella stessa riga si potrebbe creare confusione per il computer, che sarebbe incapace di distinguere fra questi ed alcuni altri tasti premuti contemporaneamente.

Il modo in cui il computer effettivamente legge la tastiera è piuttosto complicato, sebbene sia utile ed interessante capire il concetto di base dell'organizzazione della tastiera. Per nostra fortuna, non c'è bisogno di capire, o anche di usare, la routine di scansione della tastiera contenuta nella ROM: ci sono infatti due variabili molto utili che il computer tiene costantemente aggiornate nell'area delle variabili di sistema. Quest'area va dalla locazione 23552 alla locazione 23665. Diverse variabili sono lì conservate, fra cui i colori correnti, la durata del segnale acustico e, cosa per noi molto importante, il codice relativo all'ultimo tasto premuto. Leggendo, con PEEK, il valore contenuto nella locazione 23557 è possibile

sapere se è stato premuto un tasto o meno. Per provare, scrivete questo programma in BASIC:

```
   10 IF PEEK 23557=5 THEN PRINT
AT 0,0;"Tasto premuto"
  20 IF PEEK 23557<>5 THEN PRINT
 AT 0,0;"             "
  30 GO TO 10
```

Questo programma va a vedere se nella locazione 23557 è memorizzato il numero 5: se sì, scrive sul video le parole “Tasto Premuto”. Togliete il dito dal tasto ed il contenuto della locazione 23557 non sarà più uguale a 5, cosicché quelle parole saranno cancellate con degli spazi.
L’istruzione “GOTO 10” della linea 30 significa semplicemente che la routine verrà eseguita all’infinito.

CODE		CHR$	CODE		CHR$
32	=		33	=	!
34	=	"	35	=	#
36	=	$	37	=	%
38	=	&	39	=	'
40	=	(	41	=	)
42	=	*	43	=	+
44	=	,	45	=	-
46	=	.	47	=	/
48	=	0	49	=	1
50	=	2	51	=	3
52	=	4	53	=	5
54	=	6	55	=	7
56	=	8	57	=	9
58	=	:	59	=	;
60	=	<	61	=	=
62	=	>	63	=	?
64	=	@	65	=	A
66	=	B	67	=	C
68	=	D	69	=	E
70	=	F	71	=	G
72	=	H	73	=	I
74	=	J	75	=	K
76	=	L	77	=	M
78	=	N	79	=	O
80	=	P	81	=	Q
82	=	R	83	=	S
84	=	T	85	=	U
86	=	V	87	=	W
88	=	X	89	=	Y
90	=	Z	91	=	[
92	=	\	93	=	]
94	=	↑	95	=	_
96	=	£	97	=	a
98	=	b	99	=	c

CODE		CHR$	CODE		CHR$
100	=	d	101	=	e
102	=	f	103	=	g
104	=	h	105	=	i
106	=	j	107	=	k
108	=	l	109	=	m
110	=	n	111	=	o
112	=	p	113	=	q
114	=	r	115	=	s
116	=	t	117	=	u
118	=	v	119	=	w
120	=	x	121	=	y
122	=	z	123	=	{
124	=	|	125	=	}
126	=	~	127	=	©
128	=		129	=	
130	=		131	=	
132	=		133	=	
134	=		135	=	
136	=		137	=	
138	=		139	=	
140	=		141	=	
142	=		143	=	
144	=	A	145	=	B
146	=	C	147	=	D
148	=	E	149	=	F
150	=	G	151	=	H
152	=	I	153	=	J
154	=	K	155	=	L
156	=	M	157	=	N
158	=	O	159	=	P
160	=	Q	161	=	R
162	=	S	163	=	T
164	=	U			

L'informazione ottenuta da questa locazione è molto utile, ma non ci dice quale tasto è stato premuto: per saperlo si deve guardare il contenuto della locazione 23560. Questa locazione viene spesso chiamata "Last K", poiché memorizza il codice dell'ultimo tasto che à stato premuto; inoltre essa continua a tenere tale codice anche quando il tasto non è più premuto. Usando questa locazione insieme al contenuto della locazione 23557, si può vedere se un tasto è stato premuto, e di quale tasto si tratta. Ecco un breve programma di esempio in BASIC che mostra cosa succede se si usa soltanto il contenuto della locazione 23560, cioè il codice dell'ultimo tasto premuto:

```
10 PRINT PEEK 23560,: GO TO 10
```

Se lo provate, vedrete che esso è di scarsa utilità di per sé; provate adesso invece il programma che segue, nel quale si usano insieme entrame le locazioni:

```
  10 IF PEEK 23557=5 THEN PRINT
AT 0,0;CHR$ (PEEK 23560)
  20 GO TO 10
```

Si noti che se si vuole scrivere l'effettivo carattere premuto, è necessario usare l'istruzione "CHR$" sul contenuto della locazione: il computer infatti memorizza nella locazione 23560 soltanto il codice numerico corrispondente a quel carattere. Per ogni codice di carattere il computer usa un solo byte, il che permette di avere a disposizione fino ad un massimo di 256 caratteri.

Queste informazioni possono ora essere usate per scrivere una routine in linguaggio macchina che funziona in modo molto simile a quella vista prima, scritta in BASIC; questa routine restituisce il codice dell'ultimo tasto premuto memorizzando nel registro C, e che fa ritorno al BASIC soltanto se un tasto viene effettivamente premuto. Veniamo così ad usare in modo molto efficace l'istruzione di salto relativo condizionato. Però attenzione: dato che stiamo operando in linguaggio macchina, e poiché stiamo leggendo separatamente ciascun tasto, il computer non terrà conto di un'interruzione tramite il tasto "BREAK". Se non premiamo alcun tasto, rimaniamo indefinitamente in linguaggio macchina. Questa situazione può andare bene se sappiamo che un tasto dovrà per forza essere premuto, cosicché ritorneremo direttamente al BASIC e diverrà operativa l'usuale funzione del tasto BREAK. Se invece volessimo usare questo tipo di routine in un programma scritto interamente in linguaggio macchina, è consigliabile inserirvi una routine che consenta di interrompere l'esecuzione, di scrivere eventualmente un messaggio che segnali l'avvenuta interruzione e di restituire al BASIC il controllo del computer.

Vediamo ora in che modo sia possibile forzare un messaggio di errore. Lo Spectrum è in grado di fornire messaggi di errore abbastanza comprensibili, in effetti non è molto difficile usarli e generarli tramite un nostro programma, anche se essi non si adattano del tutto al linguaggio macchina. Forzare in questo modo un messaggio di errore non è difficile, specie quando si sa come fare, ma la cosa può risultare un po' confusa a questo stadio della trattazione. Usiamo invece un metodo che fa riferimento al BASIC anziché al linguaggio macchina. Forse vi ricordate che se si cerca di stampare i codici relativi ai primi 32 caratteri, si otterrà un errore: ciò è dovuto al fatto che il computer usa questi caratteri per istruzioni molto particolari, che possono riferirsi al colore oppure alla fine di una linea. Come sapete il colore viene inserito in un programma premendo il tasto "Caps Shifts", il tasto "Symbol Shift" e poi un numero. In pratica, in questo modo non si fa altro che inserire nel programma uno di questi codici, e quindi si ottiene un errore. Perché non lo proviamo? Scrivete "PRINT CHR$ (13)": questa operazione darà luogo ad un errore. È facile usare questo fatto nei nostri programmi in linguaggio macchina, caricando semplicemente nel registro C un numero minore di 32 e facendo poi ritorno al BASIC.

Il breve programma in linguaggio macchina che segue fa uso di questa possibilità per funzionare e per essere molto sintetico:

```
KEY-P EQU 23557
LASTK EQU 23560
```

```
START   LD A, (KEY-P)
        SUB 5
        JR NZ, START
BREAK?  LD A, (LAST K)
        SUB 127
        JR Z, EXIT
        LD C, (LAST K)
        LD B, Ø
        RET
EXIT    LD C, 13
        LD B, Ø
        RET
```

Il programma inizia assegnando alle due locazioni 23557 e 23560 i loro nomi di variabili di sistema, e cioè "KEY-P" e "LAST-P". In questo modo, quando nel seguito del programma si farà ancora riferimento a queste due locazioni, le si potrà chiamare con questi nomi invece che con i numeri. Ciò rende le cose più semplici per la scrittura del programma nel computer e per la successiva lettura.

La routine vera e propria inizia con il caricamento nel registro A del valore contenuto nella variabile di sistema "KEY-P", e sottraendo 5 da tale valore: se il risultato dell'operazione è zero, vorrà dire che un tasto è stato premuto. Ciò è dovuto al fatto che, come ricorderete, se si preme un tasto il valore di questa variabile viene posto a 5.

Supponendo che il risultato del registro A meno 5 sia uguale a 0, allora un tasto è stato premuto, se invece non è 0, l'istruzione di salto relativo se diverso da zero farà sì che l'esecuzione torni all'etichetta "START", per leggere di nuovo nel registro A il valore di "KEY-P".

In questo modo si entra in un anello infinito fino a che un tasto non viene effettivamente premuto. Dato che la variabile di sistema KEY-P non ci dice di quale tasto si tratta, eseguiamo ora un test per vedere se è il tasto BREAK. Ciò viene fatto quasi nello stesso modo che abbiamo usato per verificare se un tasto è stato premuto o meno: si carica nel registro A il valore della variabile di sistema LAST-K e gli si sottrae il codice del tasto BREAK. Se il tasto premuto era proprio il tasto BREAK, il risultato di questa operazione sarà 0, ed il programma salta alla breve routine di uscita. Ormai sarà chiaro che se non si tratta del tasto BREAK, ma di un qualsiasi altro tasto, si può caricare il valore di quel tasto nel registro C in modo da vederlo sul video, quando si ritorna al BASIC.

È importante azzerare il valore contenuto nel registro B perché in BASIC non si

può distinguere fra i due registri B e C, e perciò un numero diverso da zero in B farà sì che il risultato sia completamente sbagliato.
Dopo aver fatto ciò, si torna al BASIC. La parte finale del programma è una breve routine con l'etichetta "EXIT", che viene eseguita solo se viene premuto il tasto BREAK. In questo caso si carica nel registro C un valore illegale, e nel registro B il valore di 0. Il computer poi ritorna al BASIC. Come si può vedere, tutta questa routine può essere spostata a completa discrezione da un punto all'altro della memoria, e vale la pena conservarla su una cassetta per un successivo utilizzo in un programma in BASIC o in linguaggio macchina.

Ora che abbiamo visto questa procedura risulta evidente perché non è possibile di solito interrompere l'esecuzione di un programma in liguaggio macchina con il tasto BREAK, soprattutto per i programmi posti in vendita, dato che difficilmente gli autori hanno piacere di far vedere quel che hanno scritto.
Usando questo metodo vi potrete districare in un gran numero di difficoltà che potranno sorgere.

Passiamo ora a considerare il principale dispositivo di uscita del computer, il video, noto come dispositivo di uscita in quanto serve all'uscita delle informazioni contenute nel computer affinché possiamo vederle. Il video è strutturato in modo strano ma allo stesso tempo interessante: 6K byte di memoria sono usati per tenere l'effettivo disegno a punti del video, cioè per tenere i dati relativi all'accensione o allo spegnimento di ogni singolo punto dello schermo. Un altro K byte di memoria viene usato per memorizzare i colori particolari di ogni carattere e i dati relativi all'intermittenza, alla luminosità, etc. In questo volume non ci occuperemo direttamente dei colori dell'immagine che compare sul video, ma soltanto con i disegni ad alta risoluzione e con la generazione di caratteri.

Su ogni riga orizzontale dello schermo ci sono 256 punti o "pixel". Poiché essi possono soltanto essere "accesi" o "spenti", lo stato di ciascun gruppo di 8 "pixel" può essere registrato con un solo byte. Gli 8 "pixel" sono trattati come se fossero numeri binari, cioè se sono accesi vengono registrati con la cifra 1 della notazione binaria, mentre se sono spenti si usa lo 0 della notazione binaria. Il risultato, per ogni gruppo di 8 pixel è un numero binario ad otto cifre, ad esempio: "10101010".

Fate una prova, mettendo questo numero nel primo byte della memoria dello schermo con l'istruzione POKE. Usata cioè l'istruzione "POKE 16384,BIN 10101010", seguita da "Enter". Provate ora ad inserire diverse combinazioni di cifre binarie, ma assicuratevi sempre di non usare più di otto cifre, dato che ciò corrisponde al massimo al numero 255 decimale, cioè il più alto valore che il computer possa considerare. Questa struttura è valida per tutto lo schermo, ossia per 6144 volte, il che è illustrato dal seguente programma:

```
10 FOR a=16384 TO 16384+6143
20 POKE a,BIN 11111111
30 NEXT a
```

Provate ora a cambiare il valore nel numero binario e vedrete che si possono produrre effetti molto interessanti. Noterete che quando si riempie l'area di memoria dello schermo con una certa cifra, lo schermo non viene riempito in modo sequenziale, riga dopo riga, dall'alto in basso. Invece esso viene diviso in tre sezioni distinte, ciascuna delle quali viene poi suddivisa a sua volta in otto blocchi di caratteri, partendo dall'alto verso il basso. Prima viene riempita la prima riga di ogni blocco di carattere, poi la seconda riga, fino all'ottava ed ultima riga. Questo processo poi ricomincia per la sezione centrale e poi per quella inferiore. Ciò può sembrare piuttosto strano, ed in effetti lo è!!

Un altro modo per illustrare questo fatto è quello di fare un qualsiasi disegno sul video e poi conservare su nastro l'area di memoria del video, che va dall'indirizzo 16384 all'indirizzo 22527, ripulire il video e poi ricaricare lo stesso disegno. L'area di memoria dello schermo è fissa in una certa parte della memoria, e, poiché questa è sempre sparata dall'area per i programmi in BASIC o in linguaggio macchina, l'idea di caricare nel video un certo disegno all'inizio di un programma viene spesso usata come un'attraente introduzione: essa infatti non utilizza alcuna parte di memoria che potrebbe essere altrimenti usata dal programma. Scrivete il breve programma riportato qui di seguito e poi conservatelo su nastro con il comando SAVE " " CODE 16384, 6144. Ripulite il video con il comando "CLS" e poi ricaricate il video con il comando load " " code. Il risultato dovrebbe essere abbastanza interessante.

```
10 LET n=631
20 LET a=120
25 PLOT 55,27: DRAW a,a,n*PI
```

Vediamo ora un altro esempio. Caricate le due routine in linguaggio macchina per lo spostamento dei pixel verso destra e verso sinistra, e scrivete poi una breve routine che carica il disegno di video che abbiamo appena visto e lo sposta ripetutamente verso sinistra; dopo una rotazione completa di tutto il disegno, lo sposta ripetutamente verso destra. Non c'è bisogno che scriviate tale routine in linguaggio macchina. Il risultato dovrebbe dimostrare abbastanza chiaramente l'utilizzo delle routine in linguaggio macchina nell'ambito dei programmi in BASIC.

Come senza dubbio avrete immaginato, per effetto dell'insolita organizzazione dello schermo, l'esecuzione di un compito molto semplice come la stampa di un carattere sul video richiede un programma piuttosto complesso. Tuttavia possiamo concludere che, essendo il computer stesso in grado di farlo normalmente, la routine necessaria a questo scopo dovrà essere inserita in qualche punto della ROM: cercheremo dunque di trarre vantaggio da questo fatto. L'importanza di questa routine ha fatto sì che essa venisse posta molto vicino all'inizio della memoria, a partire dalla locazione 16 (decimale). Ma come accedere a tale routine? Per nostra fortuna essa è stata scritta come subroutine e può quindi essere chiamata con l'istruzione CALL. Una caratteristica di progettazione dello

Z80 è l'insita possibilità di accedere ad alcune delle routine più importanti e di più frequente utilizzo in modo veloce ed economico, e per questo scopo è stato inserito un certo gruppo di istruzioni che si riferiscono solamente ai primi otto indirizzi nella memoria. Forse la cosa più importante di tutte è che queste istruzioni sono lunghe soltanto 1 byte! Una di queste istruzioni, quella che si deve usare per accedere alla routine di stampa sul video, è "RST 16". Essa ha lo stesso effetto di "CALL16", e quindi è necessaria l'istruzione RET alla fine della routine.

Dopodiché il controllo tornerà al comando che segue RST 16.
Restano due questioni da risolvere prima di poter eseguire il comando. Anzitutto, la posizione per la stampa deve essere stabilita tramite il BASIC: anche se ciò potrebbe essere fatto dal linguaggio macchina, la cosa risulterebbe complicata senza alcuna necessità. In secondo luogo, nel registro A si dovrà caricare il codice del carattere da stampare. Ciò è utile non solo per la relativa facilità di traduzione di un carattere nel suo codice, usando la tabella fornita precedentemente, ma anche perché in linguaggio macchina è possibile utilizzare simboli grafici definiti dall'utente (User Defined Graphics).

Ecco una semplice routine per la stampa del carattere "A" nell'angolo in alto a sinistra dello schermo. Ricordatevi di accedere al linguaggio macchina nel modo indicato (in BASIC).

```
10 PRINT AT 0,0;
20 RANDOMIZE USR 30000

        org 30000
30000 3E61      ld a,97
30002 D7        rst 16
30003 C9        ret
```

Provate ora a modificare la posizione della stampa e il contenuto del registro A (il carattere da stampare). Da ciò risulta evidente anche la facilità con cui si può stampare un simbolo grafico dell'utente ("UDG"): basta caricare nel registro A il codice appropriato. È necessario a questo punto un metodo con cui un carattere UDG possa essere rappresentato in linguaggio macchina. Sfortunatamente, le routine usate in BASIC per questa operazione non sono facilmente accessibili dal linguaggio macchina. Di conseguenza, dato che ciò sarebbe in ogni caso una procedura molto complicata, vi dò io una breve routine che consente la costruzione dei caratteri UDG. Non è del tutto necessario che voi comprendiate allo stato attuale il suo modo di funzionamento: limitatevi ad usarla. Scrivetela e poi fatela eseguire con un'istruzione CALL seguita da 9 byte di dati. Il primo è il numero progressivo del simbolo grafico, per esempio il tasto A ovvero Chr$ 144=0, il tasto B ovvero Chr$ 145=1, eccetera. Ciò consente di specificare quale carattere deve essere definito. Gli altri otto byte sono le "matrici di bit" che vengono usate nel modo usuale, a formare il carattere da definire.

La prima routine che segue serve effettivamente a definire il simbolo grafico. La seconda è un esempio in cui vengono definiti i caratteri 144 e 145 (tasti A e B).

```
org 30000
equ 65368 UDG
30000 0E 00         ld c,0
30002 E1            pop hl
30003 7E            ld a,(hl)
30004 A7            and a
30005 87            add a,a
30006 87            add a,a
30007 87            add a,a
30008 11 58 FF      ld de,UDG
30011 47            ld b,a
30012 7B            ld a,e
30013 80            add a,b
30014 5F            ld e,a
30015 7A            ld a,d
30016 06 00         ld b,0
30018 88            adc a,b
30019 57            ld d,a
LOOP1
30020 79            ld a,c
30021 D6 00         sub 0
30023 30 06         jr nc,Exit
30025 0C            inc c
30026 23            inc hl
30027 7E            ld a,(hl)
30028 12            ld (de),a
30029 18 F5         jr Loop1
EXIT
30031 23            inc hl
30032 E5            push hl
30033 C9            ret

org 28000
28000 CD 30 75    call C-Gen
defb 0
defb 1
defb 3
defb 7
defb 15
defb 31
defb 63
defb 127
defb 255
28012 CD 30 75    call C-Gen
defb 1
defb 255
defb 127
defb 63
defb 31
defb 15
defb 7
defb 3
defb 1
28024 C9          ret
```

Non è necessario che voi comprendiate tutti i comandi che servono per scrivere un programma “plotter” in linguaggio macchina, ma, dato che si può rivelare molto utile, la routine che compie questa operazione è riportata qui di seguito.

```
9C40 64       LD   H,H
9C41 64       LD   H,H
9C42 3A409C   LD   A,(9C40)
9C45 CB3F     SRL  A
9C47 CB3F     SRL  A
9C49 CB3F     SRL  A
9C4B 4F       LD   C,A
9C4C 3A419C   LD   A,(9C41)
9C4F CB3F     SRL  A
9C51 CB3F     SRL  A
9C53 CB3F     SRL  A
9C55 47       LD   B,A
9C56 E600     AND  00
9C58 3E17     LD   A,17
9C5A 98       SBC  A,B
9C5B D8       RET  C
9C5C 50       LD   D,B
9C5D 47       LD   B,A
9C5E 7A       LD   A,D
9C5F CB27     SLA  A
9C61 CB27     SLA  A
9C63 CB27     SLA  A
9C65 57       LD   D,A
9C66 3A419C   LD   A,(9C41)
9C69 92       SUB  D
9C6A 57       LD   D,A
9C6B F5       PUSH AF
9C6C 78       LD   A,B
9C6D E618     AND  18
9C6F F640     OR   40
9C71 67       LD   H,A
9C72 F1       POP  AF
9C73 84       ADD  A,H
9C74 67       LD   H,A
9C75 78       LD   A,B
9C76 E607     AND  07
9C78 0F       RRCA
9C79 0F       RRCA
9C7A 0F       RRCA
9C7B 81       ADD  A,C
9C7C 6F       LD   L,A
9C7D 79       LD   A,C
9C7E CB27     SLA  A
9C80 CB27     SLA  A
9C82 CB27     SLA  A
9C84 47       LD   B,A
9C85 3A409C   LD   A,(9C40)
9C88 90       SUB  B
9C89 FE00     CP   00
9C8B 281D     JR   Z,9CAA
9C8D FE01     CP   01
```

```
9C8F 281C      JR    Z,9CAD
9C91 FE02      CP    02
9C93 281B      JR    Z,9CB0
9C95 FE03      CP    03
9C97 281A      JR    Z,9CB3
9C99 FE04      CP    04
9C9B 2819      JR    Z,9CB6
9C9D FE05      CP    05
9C9F 2818      JR    Z,9CB9
9CA1 FE06      CP    06
9CA3 2817      JR    Z,9CBC
9CA5 FE07      CP    07
9CA7 2816      JR    Z,9CBF
9CA9 C9        RET
9CAA CBFE      BIT   7,(HL)
9CAC C9        RET
9CAD CBF6      BIT   6,(HL)
9CAF C9        RET
9CB0 CBEE      BIT   5,(HL)
9CB2 C9        RET
9CB3 CBE6      BIT   4,(HL)
9CB5 C9        RET
9CB6 CBDE      BIT   3,(HL)
9CB8 C9        RET
9CB9 CBD6      BIT   2,(HL)
9CBB C9        RET
9CBC CBCE      BIT   1,(HL)
9CBE C9        RET
9CBF CBC6      BIT   0,(HL)
9CC1 C9        RET
9CC2 00        NOP
9CC3 00        NOP
9CC4 00        NOP
9CC5 00        NOP
```

CAPITOLO 10

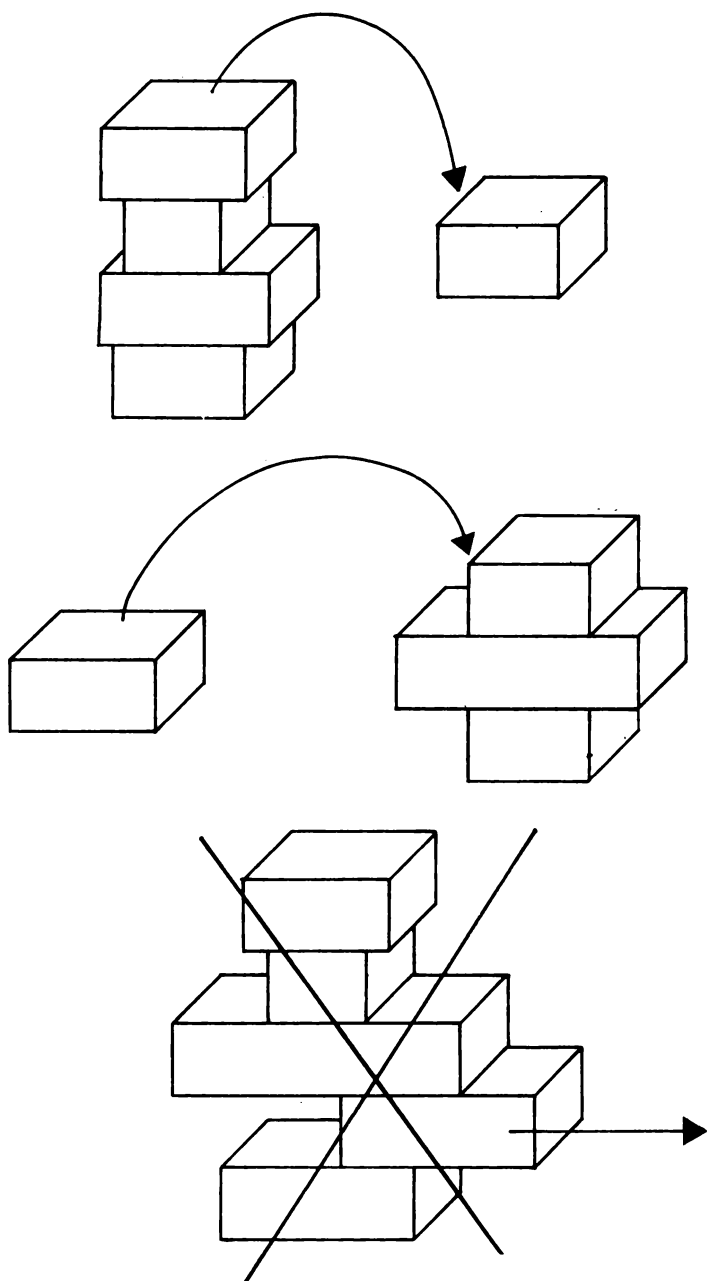
LA CATASTA

Questo capitolo sarà dedicato ad esplorare l'utilizzo di altri comandi del linguaggio macchina, alcuni nuovi concetti ed un'area particolarmente utile che viene usata dalla CPU stessa. Sarà necessario ritornare alle istruzioni "INC" e "DEC", le quali ci aiuteranno a capire le istruzioni "LDIR" e "LDDR". La piena comprensione di queste due istruzioni richiede infatti una completa familiarità con il funzionamento di "INC" e "DEC".

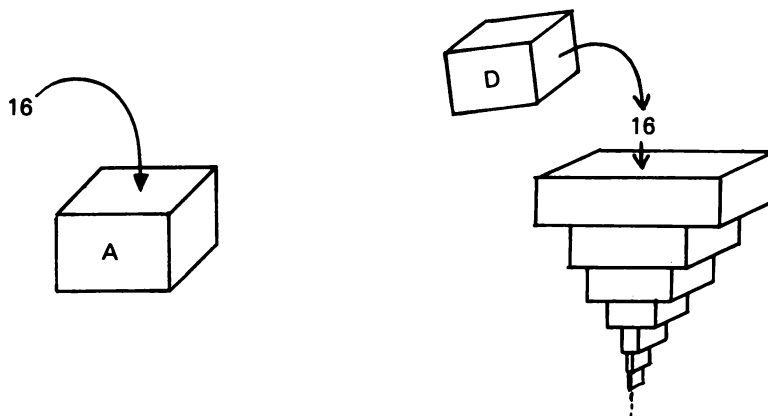
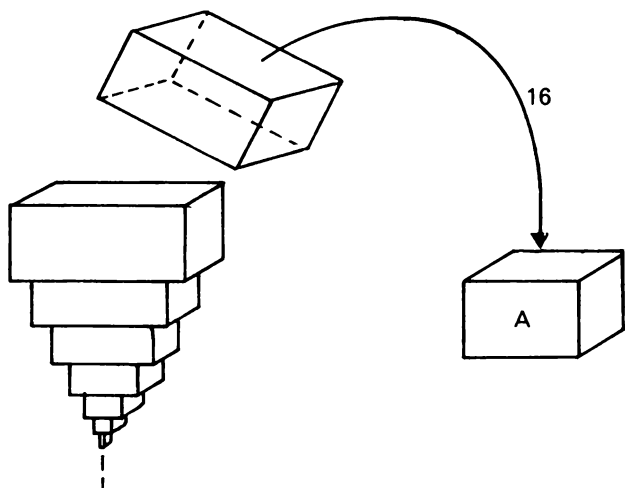
Forse per la prima volta possiamo ora usare un termine che significa esattamente ciò che rappresenta: la funzione dello "stack" (pila, catasta) è quella di essere una pila e nient'altro che una pila. Lo stack serve per accatastare dei numeri uno sull'altro. Un numero può essere posto in cima alla pila, oppure può essere rimosso, sempre dalla cima, mai da altri punti.

Possiamo immaginarci lo stack come un'alta torre di scatole: è possibile togliere la scatola che sta in cima, oppure se ne può aggiungere un'altra in cima alla torre esistente, ma ogni altra operazione la fate a vostro rischio e pericolo! Ciò è esattamente quel che succede nel linguaggio macchina, con l'unica differenza che ogni scatola è numerata con un indirizzo; tuttavia, uno dei vantaggi dello stack è che non è necessario occuparsi degli indirizzi.

Non esistono dei veri e propri limiti alle dimensioni dello stack, però quanto più esso diventa grande, tanto minore sarà lo spazio disponibile per altre cose, e inoltre c'è il rischio che si vada a scrivere sopra alcune informazioni già esistenti. La creazione di uno stack è molto facile: basta decidere a quale indirizzo nella memoria esso deve iniziare e poi assegnare quel valore alla variabile di sistema "STK-P". In questo modo si definisce il cosiddetto Puntatore di Stack: esso indica la locazione di memoria in cui si deve porre il numero successivo.

```
org 30000
equ 28000 STK-P
30000 3E 10        ld a,16
30002 32 60 6D     ld (STK-P),a
30005 21 60 6D     ld hl,STK-P
30008 35           dec (hl)
```

Senza un assembler la procedura è un po' più complicata. Anzitutto decidete la locazione da usare per tenere il puntatore di stack, poi assicuratevi che il suo valore sia 0, caricate nel registro HL quella locazione, e poi caricate nella locazione il valore del puntatore di stack.

```
org 30010
30010 3E 10           ld a,16
30012 21 60 6D        ld hl,28000
30015 77              ld (hl),a
30016 2B              dec hl
```

Quando un numero viene aggiunto in cima alla pila, il valore del puntatore di stack deve essere ridotto di 1, cosicché il successivo numero che viene aggiunto alla pila non verrà scritto allo stesso posto del precedente, ma verrà posto in una locazione con indirizzo inferiore di 1. Per mettere un numero nello stack lo si deve prima caricare nel registro A; tale registro viene poi caricato nella locazione il cui indirizzo è tenuto dal puntatore di stack, ed infine il valore del puntatore viene ridotto di 1.

A questo punto rimane solo da preparare una routine per la rimozione di un numero dello stack. In questo caso si dovrà caricare nel registro A il valore conservato in cima allo stack e poi aumentare di 1 il valore del puntatore.

Il seguente programma ci permette di creare e di usare un semplice stack. L'idea di avere una pila di numeri, da cui prenderne o aggiungerne altri, può sembrare dispersiva in termini di spazio. Invece l'utilizzo di uno stack in un programma, con l'aiuto delle routine viste prima, può risultare estremamente utile e può essere un ottimo modo per semplificare un programma. Probabilmente la maniera più semplice di usare uno stack, e che permetterà di rileggere in seguito il programma con molta facilità, è quella di chiamare "PUSH" la subroutine che mette un numero nello stack, e "POP" la subroutine che preleva un numero; si possono poi richiamare queste due subroutine con le istruzioni CALL. In questo modo si possono simulare le istruzioni vere e proprie del linguaggio macchina che normalmente servirebbero a questo scopo, ma che sono più complicate da usare.

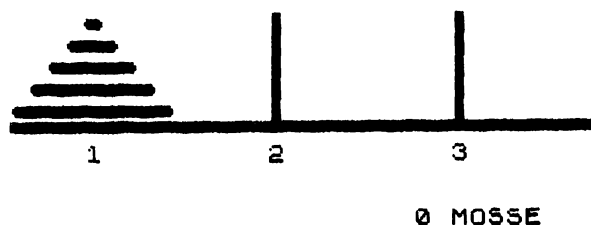
```
org 30000
equ 28000 STK-P
30000 3A 60 6D      ld a,(STK-P)
30003 21 60 6D      ld hl,STK-P
30006 23            inc hl
30007 21 60 6D      ld hl,28000
30010 7E            ld a,(hl)
30011 23            inc hl
```

Facciamo ora un breve intervallo. Il programma in BASIC che segue serve per un gioco che funziona esattamente sullo stesso principio dello stack. Anche se

non è scritto in linguaggio macchina, l'idea di base del gioco è direttamente collegata all'argomento di questo capitolo, ed in particolare al modo di servirsi dello stack. Il principio del gioco è molto vecchio e può essere che lo abbiate già incontrato altrove. Il gioco si chiama "Le Torri di Hanoi" e si tratta di spostare una pila di 5 anelli di diametro decrescente da un bastone ad un altro, ovvero da una torre all'altra. Un problema è che alla fine essi devono risultare sovrapposti esattamente come all'inizio, cioè il più largo in basso e il più stretto in cima; inoltre solo un anello può essere spostato ad ogni mossa e un anello più largo non può mai essere postato sopra uno più stretto. Ciò dovrebbe darvi un'idea dei problemi che si incontrano operando con gli stack, specialmente se si perde di vista ciò che sta in cima!

```
   2 GO SUB 2000
   5 BORDER 6: PAPER 6: INK 0: C
LS
   7 DATA "1    ▬     ","2   ▬▬▬
 ","3  ▬▬▬▬▬  ","4 ▬▬▬▬▬▬▬  ","5
▬▬▬▬▬▬▬▬▬ ","0     ▌    "
  10 RESTORE 7: LET C=10: LET M=
0: DIM A$(16,11): DIM C$(5,11):
LET W=0: DIM B$(2,1): FOR D=1 TO
 16
  20 IF D<7 THEN READ A$(D): GO
TO 40
  30 LET A$(D)=A$(6)
  40 NEXT D
 400 FOR X=1 TO 5: LET C$(X)=A$(
X): NEXT X
 470 PRINT AT 0,8;"Torri di Hano
i"
 475 PRINT AT 20,22;M;" MOSSE"
 480 FOR D=0 TO 31: PRINT AT 16,
D;"▀": NEXT D: PRINT AT 17,4; IN
K 1;"1";TAB 14;"2";TAB 24;"3"
 500 FOR B=1 TO 15: LET C=C+1: P
RINT AT C,INT (B/5-.1)*10;A$(B,2
 TO )
 510 IF C=15 THEN LET C=10
 520 NEXT B
 530 IF W=1 THEN GO TO 1600
 580 INPUT "DALLA PILA N. "; LIN
E B$(1),"ALLA PILA N. "; LINE B$
(2): PRINT AT 21,15;"
   ": IF B$(1)<STR$ 1 OR B$(1)>ST
R$ 3 OR B$(2)<STR$ 1 OR B$(2)>ST
R$ 3 THEN GO TO 1500
 600 FOR Z=1 TO 5
 610 IF A$((VAL B$(1)-1)*5+Z)<>A
$(16) THEN GO TO 640
 615 IF Z=5 AND A$((VAL B$(1)-1)
*5+Z)=A$(16) THEN GO TO 1500
 620 NEXT Z
```

```
 640 FOR Y=5 TO 1 STEP -1
 660 IF A$((VAL B$(2)-1)*5+Y)=A$
(16) THEN GO TO 1000
 680 NEXT Y
1000 IF Y=5 THEN GO TO 1010
1003 IF A$((VAL B$(2)-1)*5+Y+1)<
A$((VAL B$(1)-1)*5+Z) THEN GO TO
 1500
1010 LET A$((VAL B$(2)-1)*5+Y)=A
$((VAL B$(1)-1)*5+Z)
1020 LET A$((VAL B$(1)-1)*5+Z)=A
$(16)
1030 LET M=M+1
1040 FOR D=1 TO 5: IF A$(D+10)<>
C$(D) THEN GO TO 470
1050 NEXT D: LET W=1: GO TO 470
1500 BEEP 1,-20: PRINT AT 21,14;
 FLASH 1;"MOSSA ILLEGALE"
1520 GO TO 580
1600 PRINT AT 5,11; FLASH 1;"OTT
IMO!"; FLASH 0;''TAB 5;"Ce l'hai
 fatta in ";M;" mosse,": IF M=31
 THEN PRINT TAB 5; FLASH 1;"Non
e' possibile far meglio!": FLASH
 0: FOR x=0 TO 255: OUT 254,x: N
EXT x: GO TO 1605
1601 PRINT TAB 5;"ma puoi far me
glio"
1610 INPUT "VUOI RIPROVARE? s/n"
;Q$: IF Q$<>"S" AND Q$<>"s" THEN
 PRINT AT 21,0;"ciao...": STOP
1620 RUN
2000 RESTORE 3000
2005 FOR y=0 TO 5
2010 FOR x=0 TO 7
2020 READ chr: POKE 65368+y*8+x,
chr
2030 NEXT x
2040 NEXT y
2050 RETURN
3000 DATA 0,0,BIN 00111111,BIN 0
1111111,BIN 01111111,BIN 0011111
1,0,0
3010 DATA 0,0,255,255,255,255,0,
0
3020 DATA 0,0,BIN 11111100,BIN 1
1111110,BIN 11111110,BIN 1111110
0,0,0
3030 DATA 60,60,60,60,60,60,60,6
0
3040 DATA 0,24,60,60,60,60,60,60
3050 DATA 0,0,BIN 01111110,255,2
55,BIN 01111110,0,0
```

Torre di Hanoi

Il BASIC, direttamente o indirettamente, fa uso di tre stack distinti per le proprie operazioni: lo stack relativo alle chiamate GOSUB, che contiene le destinazioni di ritorno di tutte le subroutine scritte in BASIC; lo stack di calcolo, che viene usato per tutte le operazioni numeriche del BASIC; e infine lo stack di macchina, che è usato direttamente dalla CPU ed è accessibile tramite il linguaggio macchina. L'individuazione o l'utilizzo dello stack di macchina dal BASIC non è semplice ed è perciò consigliabile andare sempre in linguaggio macchina prima di usarlo. Prima abbiamo visto le istruzioni INC e DEC ed il modo in cui esse fanno aumentare o diminuire di 1 il valore contenuto in un registro o in una locazione. Ci sono alcuni casi in cui queste istruzioni risultano particolarmente adatte allo scopo. Uno di questi casi è il trasferimento di un intero blocco di memoria ad un'altra zona della memoria stessa. Per fare ciò useremo due puntatori, uno per ciascun'area, le quali per comodità potranno essere chiamate blocco di origine e blocco di destinazione. Le istruzioni INC e DEC ci permetteranno di aumentare o diminuire i valori dei puntatori, evitando così di dover caricare un nuovo valore dopo il trasferimento di ogni singolo byte. Per esempio:

origine

destinazione

ld a,(hl)

ld (de),a

inc hl

```
inc de
ld a,(hl)
ld (de),a
inc hl
inc de
"
"
```

Questa routine può essere soddisfacente per il trasferimento di uno o due byte, ma è indubbiamente troppo laboriosa per trasferimenti più lunghi. In questi casi useremo invece un anello, ma prima dovremo specificare, in un registro, il numero di volte che vogliamo che l'anello venga eseguito. Faremo quindi così:

origine

destinazione

numero di volte

```
loop ld a,(hl)
ld (de),a
inc de
inc hl
dec bc
ld a,b
add a,c
jr nz,loop
ret
```

Sommando il contenuto del byte più significativo B a quello del byte meno significativo C si può verificare se il contenuto di questa coppia di registri è pari a zero o no. Otteniamo così una routine abbastanza semplice per il trasferimento di un blocco di memoria.

La routine appena vista può essere ulteriormente semplificata usando due comandi che richiedono ciascuno soltanto due byte (mentre INC e DEC richiedono soltanto un byte ciascuno), e potremo così avere una routine con soltanto quattro istruzioni. I due nuovi comandi sono "LDIR" e "LDDR", che sono definiti nel modo seguente:

LDIR = "trasferimento condizionato da memoria a memoria, con incremento automatico dei registri puntatori della memoria e decremento automatico di un registro contatore di byte".

LDDR = "trasferimento condizionato da memoria a memoria, con decremento automatico dei registri puntatori della memoria e decremento automatico di un registro contatore di byte".

Ciò significa che con il comando LDIR possiamo caricare nella coppia di registri HL la locazione di inizio del blocco di origine, nei registri DE la locazione di inizio dell'area di memoria di destinazione e nei registri BC il numero di byte da trasferire. Poi, usando il comando LDIR il numero di byte specificato (in BC) viene spostato dalle locazioni HL alle locazioni DE, con i valori di HL e DE incrementati di 1, dopo il trasferimento di ogni singolo byte. Il valore di BC verrà diminuito di 1 ogni volta, fino a che non arriva a zero; a questo punto l'esecuzione del comando è conclusa ed il computer passa al comado successivo. Il comando LDDR funziona esattamente allo stesso modo, tranne ovviamente per il fatto che il contenuto dei due puntatori viene diminuito di 1 invece che incrementato. Una routine con questi comandi apparirà così:

```
LD HL, origine
LD DE, destinazione
LD BC, numero di byte
LDIR
RET
```

Per esempio, una routine per il trasferimento di tutto il contenuto del video sarà così:

```
org 25501
25501 21 00 40      ld hl,16384
25504 11 00 64      ld de,25600
25507 01 00 1C      ld bc,7168
25510 ED B0         ldir
25512 C9            ret
25513 21 00 64      ld hl,25600
25516 11 00 40      ld de,16384
25519 01 00 1C      ld bc,7168
25522 ED B0         ldir
25524 C9            ret
```

Caricate ora il programma per “disegnare”, scritto in BASIC, presentato prima, memorizzate un’immagine del video usando la prima delle due routine, ripulite quindi il video con il comando CLS, e ricopiate di nuovo l’immagine sul video con la seconda routine. Notate l’incredibile velocità di questo comando.

Siamo così giunti alla fine di questo viaggio attraverso i concetti di base della programmazione in linguaggio macchina, un viaggio che spero sia stato divertente oltre che interessante. Se siete rimasti con il desiderio di addentrarvi ancor più in questo argomento, allora ne è valsa la pena. Se poi vi sentite in condizione di utilizzare le informazioni che si possono reperire in testi più avanzati, vorrà dire che ho avuto veramente successo.

BUONA PROGRAMMAZIONE

APPENDICE A

00	0	00000000
01	1	00000001
02	2	00000010
03	3	00000011
04	4	00000100
05	5	00000101
06	6	00000110
07	7	00000111
08	8	00001000
09	9	00001001
0A	10	00001010
0B	11	00001011
0C	12	00001100
0D	13	00001101
0E	14	00001110
0F	15	00001111
10	16	00010000
11	17	00010001
12	18	00010010
13	19	00010011
14	20	00010100
15	21	00010101
16	22	00010110
17	23	00010111
18	24	00011000
19	25	00011001
1A	26	00011010
1B	27	00011011
1C	28	00011100
1D	29	00011101
1E	30	00011110
1F	31	00011111
20	32	00100000
21	33	00100001
22	34	00100010
23	35	00100011
24	36	00100100
25	37	00100101
26	38	00100110
27	39	00100111
28	40	00101000
29	41	00101001
2A	42	00101010
2B	43	00101011
2C	44	00101100
2D	45	00101101
2E	46	00101110
2F	47	00101111

30	48	00110000
31	49	00110001
32	50	00110010
33	51	00110011
34	52	00110100
35	53	00110101
36	54	00110110
37	55	00110111
38	56	00111000
39	57	00111001
3A	58	00111010
3B	59	00111011
3C	60	00111100
3D	61	00111101
3E	62	00111110
3F	63	00111111
40	64	01000000
41	65	01000001
42	66	01000010
43	67	01000011
44	68	01000100
45	69	01000101
46	70	01000110
47	71	01000111
48	72	01001000
49	73	01001001
4A	74	01001010
4B	75	01001011
4C	76	01001100
4D	77	01001101
4E	78	01001110
4F	79	01001111
50	80	01010000
51	81	01010001
52	82	01010010
53	83	01010011
54	84	01010100
55	85	01010101
56	86	01010110
57	87	01010111
58	88	01011000
59	89	01011001
5A	90	01011010
5B	91	01011011
5C	92	01011100
5D	93	01011101
5E	94	01011110
5F	95	01011111
60	96	01100000
61	97	01100001
62	98	01100010
63	99	01100011
64	100	01100100
65	101	01100101
66	102	01100110
67	103	01100111
68	104	01101000
69	105	01101001

6A	106	01101010
6B	107	01101011
6C	108	01101100
6D	109	01101101
6E	110	01101110
6F	111	01101111
70	112	01110000
71	113	01110001
72	114	01110010
73	115	01110011
74	116	01110100
75	117	01110101
76	118	01110110
77	119	01110111
78	120	01111000
79	121	01111001
7A	122	01111010
7B	123	01111011
7C	124	01111100
7D	125	01111101
7E	126	01111110
7F	127	01111111
80	128	10000000
81	129	10000001
82	130	10000010
83	131	10000011
84	132	10000100
85	133	10000101
86	134	10000110
87	135	10000111
88	136	10001000
89	137	10001001
8A	138	10001010
8B	139	10001011
8C	140	10001100
8D	141	10001101
8E	142	10001110
8F	143	10001111
90	144	10010000
91	145	10010001
92	146	10010010
93	147	10010011
94	148	10010100
95	149	10010101
96	150	10010110
97	151	10010111
98	152	10011000
99	153	10011001
9A	154	10011010
9B	155	10011011
9C	156	10011100
9D	157	10011101
9E	158	10011110
9F	159	10011111
A0	160	10100000
A1	161	10100001
A2	162	10100010
A3	163	10100011

A4	164	10100100
A5	165	10100101
A6	166	10100110
A7	167	10100111
A8	168	10101000
A9	169	10101001
AA	170	10101010
AB	171	10101011
AC	172	10101100
AD	173	10101101
AE	174	10101110
AF	175	10101111
B0	176	10110000
B1	177	10110001
B2	178	10110010
B3	179	10110011
B4	180	10110100
B5	181	10110101
B6	182	10110110
B7	183	10110111
B8	184	10111000
B9	185	10111001
BA	186	10111010
BB	187	10111011
BC	188	10111100
BD	189	10111101
BE	190	10111110
BF	191	10111111
C0	192	11000000
C1	193	11000001
C2	194	11000010
C3	195	11000011
C4	196	11000100
C5	197	11000101
C6	198	11000110
C7	199	11000111
C8	200	11001000
C9	201	11001001
CA	202	11001010
CB	203	11001011
CC	204	11001100
CD	205	11001101
CE	206	11001110
CF	207	11001111
D0	208	11010000
D1	209	11010001
D2	210	11010010
D3	211	11010011
D4	212	11010100
D5	213	11010101
D6	214	11010110
D7	215	11010111
D8	216	11011000
D9	217	11011001
DA	218	11011010
DB	219	11011011
DC	220	11011100
DD	221	11011101

DE	222	11011110
DF	223	11011111
E0	224	11100000
E1	225	11100001
E2	226	11100010
E3	227	11100011
E4	228	11100100
E5	229	11100101
E6	230	11100110
E7	231	11100111
E8	232	11101000
E9	233	11101001
EA	234	11101010
EB	235	11101011
EC	236	11101100
ED	237	11101101
EE	238	11101110
EF	239	11101111
F0	240	11110000
F1	241	11110001
F2	242	11110010
F3	243	11110011
F4	244	11110100
F5	245	11110101
F6	246	11110110
F7	247	11110111
F8	248	11111000
F9	249	11111001
FA	250	11111010
FB	251	11111011
FC	252	11111100
FD	253	11111101
FE	254	11111110
FF	255	11111111

```
  10 FOR a=0 TO 255
  20 GO SUB 400: LPRINT "      "
;a;"        ";: GO SUB 200: LPRINT
  30 NEXT a: STOP
 200 LET n=a
 210 FOR x=7 TO 0 STEP -1
 220 IF INT (n/(2↑x))=1 THEN GO
TO 300
 230 LPRINT 0;
 240 NEXT x
 250 RETURN
 300 LPRINT 1;
 310 LET n=n-2↑x
 320 GO TO 240
 400 LET n=a
 410 LET x=INT (n/16)
 420 LET y=((n/16)-INT (n/16))*1
6
 430 IF x>9 THEN LET x=x+7
 440 IF y>9 THEN LET y=y+7
 450 LET x=x+48: LET y=y+48
 460 LPRINT CHR$ x;CHR$ y;
 470 RETURN
```

APPENDICE B

CODICI OPERATIVI DELLO Z80, ORDINATI SECONDO IL MNEMONICO

Codice		Esadecimale	Decimale
ADC	A,(HL)	8E	142
ADC	A,(IX+d)	DD8Edd	221,142,XX
ADC	A,(IY+d)	FD8Edd	253,142,XX
ADC	A,A	8F	143
ADC	A,B	88	136
ADC	A,C	89	137
ADC	A,D	8A	138
ADC	A,E	8B	139
ADC	A,H	8C	140
ADC	A,L	8D	141
ADC	A,xx	CExx	206,XX
ADC	HL,BC	ED4A	237,74
ADC	HL,DE	ED5A	237,90
ADC	HL,HL	ED6A	237,106
ADC	HL,SP	ED7A	237,122
ADD	A,(HL)	86	134
ADD	A,(IX+d)	DD86dd	221,134,XX
ADD	A,(IY+d)	FD86dd	253,134,XX
ADD	A,A	87	135
ADD	A,B	80	128
ADD	A,C	81	129
ADD	A,D	82	130
ADD	A,E	83	131
ADD	A,H	84	132
ADD	A,L	85	133
ADD	A,xx	C6xx	198,XX
ADD	HL,BC	09	9
ADD	HL,DE	19	25
ADD	HL,HL	29	41
ADD	HL,SP	39	57
ADD	IX,BC	DD09	221,9

Codice		Esadecimale	Decimale
ADD	IX,DE	DD19	221,25
ADD	IX,IX	DD29	221,41
ADD	IX,SP	DD39	221,57
ADD	IY,BC	FD09	253,9
ADD	IY,DE	FD19	253,25
ADD	IY,IY	FD29	253,41
ADD	IY,SP	FD39	253,57
AND	(HL)	A6	166
AND	(IX+d)	DDA6dd	221,166,XX
AND	(IY+d)	FDA6dd	253,166,XX
AND	A	A7	167
AND	B	A0	160
AND	C	A1	161
AND	D	A2	162
AND	E	A3	163
AND	H	A4	164
AND	L	A5	165
AND	xx	E6xx	230,XX
BIT	0,(HL)	CB46	203,70
BIT	0,(IX+d)	DDCBdd46	221,203,XX,70
BIT	0,(IY+d)	FDCBdd46	253,203,XX,70
BIT	0,A	CB47	203,71
BIT	0,B	CB40	203,64
BIT	0,C	CB41	203,65
BIT	0,D	CB42	203,66
BIT	0,E	CB43	203,67
BIT	0,H	CB44	203,68
BIT	0,L	CB45	203,69
BIT	1,(HL)	CB4E	203,78
BIT	1,(IX+d)	DDCBdd4E	221,203,XX,78
BIT	1,(IY+d)	FDCBdd4E	253,203,XX,78
BIT	1,A	CB4F	203,79
BIT	1,B	CB48	203,72
BIT	1,C	CB49	203,73
BIT	1,D	CB4A	203,74
BIT	1,E	CB4B	203,75
BIT	1,H	CB4C	203,76
BIT	1,L	CB4D	203,77
BIT	2,(HL)	CB56	203,86
BIT	2,(IX+d)	DDCBdd56	221,203,XX,86
BIT	2,(IY+d)	FDCBdd56	253,203,XX,86
BIT	2,A	CB57	203,87

Codice	Esadecimale	Decimale
BIT 2,B	CB50	203,80
BIT 2,C	CB51	203,81
BIT 2,D	CB52	203,82
BIT 2,E	CB53	203,83
BIT 2,H	CB54	203,84
BIT 2,L	CB55	203,85
BIT 3,(HL)	CB5E	203,94
BIT 3,(IX+d)	DDCBdd5E	221,203,XX,94
BIT 3,(IY+d)	FDCBdd5E	253,203,XX,94
BIT 3,A	CB5F	203,95
BIT 3,B	CB58	203,88
BIT 3,C	CB59	203,89
BIT 3,D	CB5A	203,90
BIT 3,E	CB5B	203,91
BIT 3,H	CB5C	203,92
BIT 3,L	CB5D	203,93
BIT 4,(HL)	CB66	203,102
BIT 4,(IX+d)	DDCBdd66	221,203,XX,102
BIT 4,(IY+d)	FDCBdd66	253,203,XX,102
BIT 4,A	CB67	203,103
BIT 4,B	CB60	203,96
BIT 4,C	CB61	203,97
BIT 4,D	CB62	203,98
BIT 4,E	CB63	203,99
BIT 4,H	CB64	203,100
BIT 4,L	CB65	203,101
BIT 5,(HL)	CB6E	203,110
BIT 5,(IX+d)	DDCBdd6E	221,203,XX,110
BIT 5,(IY+d)	FDCBdd6E	253,203,XX,110
BIT 5,A	CB6F	203,111
BIT 5,B	CB68	203,104
BIT 5,C	CB69	203,105
BIT 5,D	CB6A	203,106
BIT 5,E	CB6B	203,107
BIT 5,H	CB6C	203,108
BIT 5,L	CB6D	203,109
BIT 6,(HL)	CB76	203,118
BIT 6,(IX+d)	DDCBdd76	221,203,XX,118
BIT 6,(IY+d)	FDCBdd76	253,203,XX,118
BIT 6,A	CB77	203,119
BIT 6,B	CB70	203,112
BIT 6,C	CB71	203,113

Codice	Esadecimale	Decimale
BIT 6,D	CB72	203,114
BIT 6,E	CB73	203,115
BIT 6,H	CB74	203,116
BIT 6,L	CB75	203,117
BIT 7,(HL)	CB7E	203,126
BIT 7,(IX+d)	DDCBdd7E	221,203,XX,126
BIT 7,(IY+d)	FDCBdd7E	253,203,XX,126
BIT 7,A	CB7F	203,127
BIT 7,B	CB78	203,120
BIT 7,C	CB79	203,121
BIT 7,D	CB7A	203,122
BIT 7,E	CB7B	203,123
BIT 7,H	CB7C	203,124
BIT 7,L	CB7D	203,125
CALL C,xxxx	DCxxxx	220,XX,XX
CALL M,xxxx	FCxxxx	252,XX,XX
CALL NC,xxxx	D4xxxx	212,XX,XX
CALL NZ,xxxx	C4xxxx	196,XX,XX
CALL P,xxxx	F4xxxx	244,XX,XX
CALL PE,xxxx	ECxxxx	236,XX,XX
CALL PO,xxxx	E4xxxx	228,XX,XX
CALL xxxx	CDxxxx	205,XX,XX
CALL Z,xxxx	CCxxxx	204,XX,XX
CCF	3F	63
CP (HL)	BE	190
CP (IX+d)	DDBEdd	221,190,XX
CP (IY+d)	FDBEdd	253,190,XX
CP A	BF	191
CP B	B8	184
CP C	B9	185
CP D	BA	186
CP E	BB	187
CP H	BC	188
CP L	BD	189
CP xx	FExx	254,XX
CPD	EDA9	237,169
CPDR	EDB9	237,185
CPI	EDA1	237,161
CPIR	EDB1	237,177
CPL	2F	47
DAA	27	39
DEC (HL)	35	53

Codice		Esadecimale	Decimale
DEC	(IX+d)	DD35dd	221,53,XX
DEC	(IY+d)	FD35dd	253,53,XX
DEC	A	3D	61
DEC	B	05	5
DEC	BC	0B	11
DEC	C	0D	13
DEC	D	15	21
DEC	DE	1B	27
DEC	E	1D	29
DEC	H	25	37
DEC	HL	2B	43
DEC	IX	DD2B	221,43
DEC	IY	FD2B	253,43
DEC	L	2D	45
DEC	SP	3B	59
DI		F3	243
DJNZ	xx	10xx	16,XX
EI		FB	251
EX	(SP),HL	E3	227
EX	(SP),IX	DDE3	221,227
EX	(SP),IY	FDE3	253,227
EX	AF,AF	08	8
EX	DE,HL	EB	235
Exx		D9	217
HALT		76	118
IM	0	ED46	237,70
IM	1	ED56	237,86
IM	2	ED5E	237,94
IN	A,(C)	ED78	237,120
IN	A,(xx)	DBxx	219,XX
IN	B,(C)	ED40	237,64
IN	C,(C)	ED48	237,72
IN	D,(C)	ED50	237,80
IN	E,(C)	ED58	237,88
IN	H,(C)	ED60	237,96
IN	L,(C)	ED68	237,104
INC	(HL)	34	52
INC	(IX+d)	DD34dd	221,52,XX
INC	(IY+d)	FD34dd	253,52,XX
INC	A	3C	60
INC	B	04	4
INC	BC	03	3

Codice		Esadecimale	Decimale
INC	C	0C	12
INC	D	14	20
INC	DE	13	19
INC	E	1C	28
INC	H	24	36
INC	HL	23	35
INC	IX	DD23	221,35
INC	IY	FD23	253,35
INC	L	2C	44
INC	SP	33	51
IND		EDAA	237,170
INDR		EDBA	237,186
INI		EDA2	237,162
INIR		EDB2	237,178
JP	(HL)	E9	233
JP	(IX)	DDE9	221,233
JP	(IY)	FDE9	253,233
JP	C,xxxx	DAxxxx	218,XX,XX
JP	M,xxxx	FAxxxx	250,XX,XX
JP	NC,xxxx	D2xxxx	210,XX,XX
JP	NZ,xxxx	C2xxxx	194,XX,XX
JP	P,xxxx	F2xxxx	242,XX,XX
JP	PE,xxxx	EAxxxx	234,XX,XX
JP	PO,xxxx	E2xxxx	226,XX,XX
JP	xxxx	C3xxxx	195,XX,XX
JP	Z,xxxx	CAxxxx	202,XX,XX
JR	C,xx	38xx	56,XX
JR	NC,xx	30xx	48,XX
JR	NZ,xx	20xx	32,XX
JR	xx	18xx	24,XX
JR	Z,xx	28xx	40,XX
LD	(BC),A	02	2
LD	(DE),A	12	18
LD	HL,(xxxx)	2Axxxx	42,XX,XX
LD	(HL),A	77	119
LD	(HL),B	70	112
LD	(HL),C	71	113
LD	(HL),D	72	114
LD	(HL),E	73	115
LD	(HL),H	74	116
LD	(HL),L	75	117
LD	(HL),xx	36xx	54,XX

Codice		Esadecimale	Decimale
LD	(IX+d),A	DD77dd	221,119,XX
LD	(IX+d),B	DD70dd	221,112,XX
LD	(IX+d),C	DD71dd	221,113,XX
LD	(IX+d),D	DD72dd	221,114,XX
LD	(IX+d),E	DD73dd	221,115,XX
LD	(IX+d),H	DD74dd	221,116,XX
LD	(IX+d),L	DD75dd	221,117,XX
LD	(IX+d),xx	DD36ddxx	221,54,XX,XX
LD	(IY+d),A	FD77dd	253,119,XX
LD	(IY+d),B	FD70dd	253,112,XX
LD	(IY+d),C	FD71dd	253,113,XX
LD	(IY+d),D	FD72dd	253,114,XX
LD	(IY+d),E	FD73dd	253,115,XX
LD	(IY+d),H	FD74dd	253,116,XX
LD	(IY+d),L	FD75dd	253,117,XX
LD	(IY+d),xx	FD36ddxx	253,54,XX,XX
LD	(xxxx),A	32xxxx	50,XX,XX
LD	(xxxx),BC	ED43xxxx	237,67,XX,XX
LD	(xxxx),DE	ED53xxxx	237,83,XX,XX
LD	(xxxx),HL	22xxxx	34,XX,XX
LD	(xxxx),IX	DD22xxxx	221,34,XX,XX
LD	(xxxx),IY	FD22xxxx	253,34,XX,XX
LD	(xxxx),SP	ED73xxxx	237,115,XX,XX
LD	A,(BC)	0A	10
LD	A,(DE)	1A	26
LD	A,(HL)	7E	126
LD	A,(IX+d)	DD7Edd	221,126,XX
LD	A,(IY+d)	FD7Edd	253,126,XX
LD	A,(xxxx)	3Axxxx	58,XX,XX
LD	A,A	7F	127
LD	A,B	78	120
LD	A,C	79	121
LD	A,D	7A	122
LD	A,E	7B	123
LD	A,H	7C	124
LD	A,I	ED57	237,87
LD	A,L	7D	125
LD	A,R	ED5F	237,95
LD	A,xx	3Exx	62,XX
LD	B,(HL)	46	70
LD	B,(IX+d)	DD46dd	221,70,XX
LD	B,(IY+d)	FD46dd	253,70,XX

Codice		Esadecimale	Decimale
LD	B,A	47	71
LD	B,B	40	64
LD	B,C	41	65
LD	B,D	42	66
LD	B,E	43	67
LD	B,H	44	68
LD	B,L	45	69
LD	B,xx	06xx	6,XX
LD	BC,(xxxx)	ED4Bxxxx	237,75,XX,XX
LD	BC,xxxx	01xxxx	1,XX,XX
LD	C,(HL)	4E	78
LD	C,(IX+d)	DD4Edd	221,78,XX
LD	C,(IY+d)	FD4Edd	253,78,XX
LD	C,A	4F	79
LD	C,B	48	72
LD	C,C	49	73
LD	C,D	4A	74
LD	C,E	4B	75
LD	C,H	4C	76
LD	C,L	4D	77
LD	C,xx	0Exx	14,XX
LD	D,(HL)	56	86
LD	D,(IX+d)	DD56dd	221,86,XX
LD	D,(IY+d)	FD56dd	253,86,XX
LD	D,A	57	87
LD	D,B	50	80
LD	D,C	51	81
LD	D,D	52	82
LD	D,E	53	83
LD	D,H	54	84
LD	D,L	55	85
LD	D,xx	16xx	22,XX
LD	DE,(xxxx)	ED5Bxxxx	237,91,XX,XX
LD	DE,xxxx	11xxxx	17,XX,XX
LD	E,(HL)	5E	94
LD	E,(IX+d)	DD5Edd	221,94,XX
LD	E,(IY+d)	FD5Edd	253,94,XX
LD	E,A	5F	95
LD	E,B	58	88
LD	E,C	59	89
LD	E,D	5A	90
LD	E,E	5B	91

Codice		Esadecimale	Decimale
LD	E,H	5C	92
LD	E,L	5D	93
LD	E,xx	1Exx	30,XX
LD	H,(HL)	66	102
LD	H,(IX+d)	DD66dd	221,102,XX
LD	H,(IY+d)	FD66dd	253,102,XX
LD	H,A	67	103
LD	H,B	60	96
LD	H,C	61	97
LD	H,D	62	98
LD	H,E	63	99
LD	H,H	64	100
LD	H,L	65	101
LD	H,xx	26xx	38,XX
LD	HL,xxxx	21xxxx	33,XX,XX
LD	I,A	ED47	237,71
LD	IX,(xxxx)	DD2Axxxx	221,42,XX,XX
LD	IX,xxxx	DD21xxxx	221,33,XX,XX
LD	IY,(xxxx)	FD2Axxxx	253,42,XX,XX
LD	IY,xxxx	FD21xxxx	253,33,XX,XX
LD	L,(HL)	6E	110
LD	L,(IX+d)	DD6Edd	221,110,XX
LD	L,(IY+d)	FD6Edd	253,110,XX
LD	L,A	6F	111
LD	L,B	68	104
LD	L,C	69	105
LD	L,D	6A	106
LD	L,E	6B	107
LD	L,H	6C	108
LD	L,L	6D	109
LD	L,xx	2Exx	46,XX
LD	R,A	ED4F	237,79
LD	SP,(xxxx)	ED7Bxxxx	237,123,XX,XX
LD	SP,HL	F9	249
LD	SP,IX	DDF9	221,249
LD	SP,IY	FDF9	253,249
LD	SP,xxxx	31xxxx	49,XX,XX
LDD		EDA8	237,168
LDDR		EDB8	237,184
LDI		EDA0	237,160
LDIR		EDB0	237,176
NEG		ED44	237,68

Codice		Esadecimale	Decimale
NOP		00	0
OR	(HL)	B6	182
OR	(IX+d)	DDB6dd	221,182,XX
OR	(IY+d)	FDB6dd	253,182,XX
OR	A	B7	183
OR	B	B0	176
OR	C	B1	177
OR	D	B2	178
OR	E	B3	179
OR	H	B4	180
OR	L	B5	181
OR	xx	F6xx	246,XX
OTDR		EDBB	237,187
OTIR		EDB3	237,179
OUT	(C),A	ED79	237,121
OUT	(C),B	ED41	237,65
OUT	(C),C	ED49	237,73
OUT	(C),D	ED51	237,81
OUT	(C),E	ED59	237,89
OUT	(C),H	ED61	237,97
OUT	(C),L	ED69	237,105
OUT	(xx),A	D3xx	211,XX
OUTD		EDAB	237,171
OUTI		EDA3	237,163
POP	AF	F1	241
POP	BC	C1	193
POP	DE	D1	209
POP	HL	E1	225
POP	IX	DDE1	221,225
POP	IY	FDE1	253,225
PUSH	AF	F5	245
PUSH	BC	C5	197
PUSH	DE	D5	213
PUSH	HL	E5	229
PUSH	IX	DDE5	221,229
PUSH	IY	FDE5	253,229
RES	0,(HL)	CB86	203,134
RES	0,(IX+d)	DDCBdd86	221,203,XX,134
RES	0,(IY+d)	FDCBdd86	253,203,XX,134
RES	0,A	CB87	203,135
RES	0,B	CB80	203,128
RES	0,C	CB81	203,129

Codice		Esadecimale	Decimale
RES	0,D	CB82	203,130
RES	0,E	CB83	203,131
RES	0,H	CB84	203,132
RES	0,L	CB85	203,133
RES	1,(HL)	CB8E	203,142
RES	1,(IX+d)	DDCBdd8E	221,203,XX,142
RES	1,(IY+d)	FDCBdd8E	253,203,XX,142
RES	1,A	CB8F	203,143
RES	1,B	CB88	203,136
RES	1,C	CB89	203,137
RES	1,D	CB8A	203,138
RES	1,E	CB8B	203,139
RES	1,H	CB8C	203,140
RES	1,L	CB8D	203,141
RES	2,(HL)	CB96	203,150
RES	2,(IX+d)	DDCBdd96	221,203,XX,150
RES	2,(IY+d)	FDCBdd96	253,203,XX,150
RES	2,A	CB97	203,151
RES	2,B	CB90	203,144
RES	2,C	CB91	203,145
RES	2,D	CB92	203,146
RES	2,E	CB93	203,147
RES	2,H	CB94	203,148
RES	2,L	CB95	203,149
RES	3,(HL)	CB9E	203,158
RES	3,(IX+d)	DDCBdd9E	221,203,XX,158
RES	3,(IY+d)	FDCBdd9E	253,203,XX,158
RES	3,A	CB9F	203,159
RES	3,B	CB98	203,152
RES	3,C	CB99	203,153
RES	3,D	CB9A	203,154
RES	3,E	CB9B	203,155
RES	3,H	CB9C	203,156
RES	3,L	CB9D	203,157
RES	4,(HL)	CBA6	203,166
RES	4,(IX+d)	DDCBddA6	221,203,XX,166
RES	4,(IY+d)	FDCBddA6	253,203,XX,166
RES	4,A	CBA7	203,167
RES	4,B	CBA0	203,160
RES	4,C	CBA1	203,161
RES	4,D	CBA2	203,162
RES	4,E	CBA3	203,163

Codice		Esadecimale	Decimale
RES	4,H	CBA4	203,164
RES	4,L	CBA5	203,165
RES	5,(HL)	CBAE	203,174
RES	5,(IX+d)	DDCBddAE	221,203,XX,174
RES	5,(IY+d)	FDCBddAE	253,203,XX,174
RES	5,A	CBAF	203,175
RES	5,B	CBA8	203,168
RES	5,C	CBA9	203,169
RES	5,D	CBAA	203,170
RES	5,E	CBAB	203,171
RES	5,H	CBAC	203,172
RES	5,L	CBAD	203,173
RES	6,(HL)	CBB6	203,182
RES	6,(IX+d)	DDCBddB6	221,203,XX,182
RES	6,(IY+d)	FDCBddB6	253,203,XX,182
RES	6,A	CBB7	203,183
RES	6,B	CBB0	203,176
RES	6,C	CBB1	203,177
RES	6,D	CBB2	203,178
RES	6,E	CBB3	203,179
RES	6,H	CBB4	203,180
RES	6,L	CBB5	203,181
RES	7,(HL)	CBBE	203,190
RES	7,(IX+d)	DDCBddBE	221,203,XX,190
RES	7,(IY+d)	FDCBddBE	253,203,XX,190
RES	7,A	CBBF	203,191
RES	7,B	CBB8	203,184
RES	7,C	CBB9	203,185
RES	7,D	CBBA	203,186
RES	7,E	CBBB	203,187
RES	7,H	CBBC	203,188
RES	7,L	CBBD	203,189
RET		C9	201
RET	C	D8	216
RET	M	F8	248
RET	NC	D0	208
RET	NZ	C0	192
RET	P	F0	240
RET	PE	E8	232
RET	PO	E0	224
RET	Z	C8	200
RETI		ED4D	237,77

Codice		Esadecimale	Decimale
RETN		ED45	237,69
RL	(HL)	CB16	203,22
RL	(IX+d)	DDCBdd16	221,203,XX,22
RL	(IY+d)	FDCBdd16	253,203,XX,22
RL	A	CB17	203,23
RL	B	CB10	203,16
RL	C	CB11	203,17
RL	D	CB12	203,18
RL	E	CB13	203,19
RL	H	CB14	203,20
RL	L	CB15	203,21
RLA		17	23
RLC	(HL)	CB06	203,6
RLC	(IX+d)	DDCBdd06	221,203,XX,6
RLC	(IY+d)	FDCBdd06	253,203,XX,6
RLC	A	CB07	203,7
RLC	B	CB00	203,0
RLC	C	CB01	203,1
RLC	D	CB02	203,2
RLC	E	CB03	203,3
RLC	H	CB04	203,4
RLC	L	CB05	203,5
RLCA		07	7
RLD		ED6F	237,111
RR	(HL)	CB1E	203,30
RR	(IX+d)	DDCBdd1E	221,203,XX,30
RR	(IY+d)	FDCBdd1E	253,203,XX,30
RR	A	CB1F	203,31
RR	B	CB18	203,24
RR	C	CB19	203,25
RR	D	CB1A	203,26
RR	E	CB1B	203,27
RR	H	CB1C	203,28
RR	L	CB1D	203,29
RRA		1F	31
RRC	(HL)	CB0E	203,14
RRC	(IX+d)	DDCBdd0E	221,203,XX,14
RRC	(IY+d)	FDCBdd0E	253,203,XX,14
RRC	A	CB0F	203,15
RRC	B	CB08	203,8
RRC	C	CB09	203,9
RRC	D	CB0A	203,10

Codice		Esadecimale	Decimale
RRC	E	CB0B	203,11
RRC	H	CB0C	203,12
RRC	L	CB0D	203,13
RRCA		0F	15
RRD		ED67	237,103
RST	0	C7	199
RST	10h	D7	215
RST	18h	DF	223
RST	20h	E7	231
RST	28h	EF	239
RST	30h	F7	247
RST	38h	FF	255
RST	8	CF	207
SBC	A,(HL)	9E	158
SBC	A,(IX+d)	DD9Edd	221,158,XX
SBC	A,(IY+d)	FD9Edd	253,158,XX
SBC	A,A	9F	159
SBC	A,B	98	152
SBC	A,C	99	153
SBC	A,D	9A	154
SBC	A,E	9B	155
SBC	A,H	9C	156
SBC	A,L	9D	157
SBC	A,xx	DExx	222,XX
SBC	HL,BC	ED42	237,66
SBC	HL,DE	ED52	237,82
SBC	HL,HL	ED62	237,98
SBC	HL,SP	ED72	237,114
SCF		37	55
SET	0,(HL)	CBC6	203,198
SET	0,(IX+d)	DDCBddC6	221,203,XX,198
SET	0,(IY+d)	FDCBddC6	253,203,XX,198
SET	0,A	CBC7	203,199
SET	0,B	CBC0	203,192
SET	0,C	CBC1	203,193
SET	0,D	CBC2	203,194
SET	0,E	CBC3	203,195
SET	0,H	CBC4	203,196
SET	0,L	CBC5	203,197
SET	1,(HL)	CBCE	203;206
SET	1,(IX+d)	DDCBddCE	221,203,XX,206
SET	1,(IY+d)	FDCBddCE	253,203,XX,206

Codice	Esadecimale	Decimale
SET 1,A	CBCF	203,207
SET 1,B	CBC8	203,200
SET 1,C	CBC9	203,201
SET 1,D	CBCA	203,202
SET 1,E	CBCB	203,203
SET 1,H	CBCC	203,204
SET 1,L	CBCD	203,205
SET 2,(HL)	CBD6	203,214
SET 2,(IX+d)	DDCBddD6	221,203,XX,214
SET 2,(IY+d)	FDCBddD6	253,203,XX,214
SET 2,A	CBD7	203,215
SET 2,B	CBD0	203,208
SET 2,C	CBD1	203,209
SET 2,D	CBD2	203,210
SET 2,E	CBD3	203,211
SET 2,H	CBD4	203,212
SET 2,L	CBD5	203,213
SET 3,(HL)	CBDE	203,222
SET 3,(IX+d)	DDCBddDE	221,203,XX,222
SET 3,(IY+d)	FDCBddDE	253,203,XX,222
SET 3,A	CBDF	203,223
SET 3,B	CBD8	203,216
SET 3,C	CBD9	203,217
SET 3,D	CBDA	203,218
SET 3,E	CBDB	203,219
SET 3,H	CBDC	203,220
SET 3,L	CBDD	203,221
SET 4,(HL)	CBE6	203,230
SET 4,(IX+d)	DDCBddE6	221,203,XX,230
SET 4,(IY+d)	FDCBddE6	253,203,XX,230
SET 4,A	CBE7	203,231
SET 4,B	CBE0	203,224
SET 4,C	CBE1	203,225
SET 4,D	CBE2	203,226
SET 4,E	CBE3	203,227
SET 4,H	CBE4	203,228
SET 4,L	CBE5	203,229
SET 5,(HL)	CBEE	203,238
SET 5,(IX+d)	DDCBddEE	221,203,XX,238
SET 5,(IY+d)	FDCBddEE	253,203,XX,238
SET 5,A	CBEF	203,239
SET 5,B	CBE8	203,232

Codice	Esadecimale	Decimale
SET 5,C	CBE9	203,233
SET 5,D	CBEA	203,234
SET 5,E	CBEB	203,235
SET 5,H	CBEC	203,236
SET 5,L	CBED	203,237
SET 6,(HL)	CBF6	203,246
SET 6,(IX+d)	DDCBddF6	221,203,XX,246
SET 6,(IY+d)	FDCBddF6	253,203,XX,246
SET 6,A	CBF7	203,247
SET 6,B	CBF0	203,240
SET 6,C	CBF1	203,241
SET 6,D	CBF2	203,242
SET 6,E	CBF3	203,243
SET 6,H	CBF4	203,244
SET 6,L	CBF5	203,245
SET 7,(HL)	CBFE	203,254
SET 7,(IX+d)	DDCBffFE	221,203,XX,254
SET 7,(IY+d)	FDCBddFE	253,203,XX,254
SET 7,A	CBFF	203,255
SET 7,B	CBF8	203,248
SET 7,C	CBF9	203,249
SET 7,D	CBFA	203,250
SET 7,E	CBFB	203,251
SET 7,H	CBFC	203,252
SET 7,L	CBFD	203,253
SLA (HL)	CB26	203,38
SLA (IX+d)	DDCBdd26	221,203,XX,38
SLA (IY+d)	FDCBdd26	253,203,XX,38
SLA A	CB27	203,39
SLA B	CB20	203,32
SLA C	CB21	203,33
SLA D	CB22	203,34
SLA E	CB23	203,35
SLA H	CB24	203,36
SLA L	CB25	203,37
SRA (HL)	CB2E	203,46
SRA (IX+d)	DDCBdd2E	221,203,XX,46
SRA (IY+d)	FDCBdd2E	253,203,XX,46
SRA A	CB2F	203,47
SRA B	CB28	203,40
SRA C	CB29	203,41
SRA D	CB2A	203,42

Codice		Esadecimale	Decimale
SRA	E	CB2B	203,43
SRA	H	CB2C	203,44
SRA	L	CB2D	203,45
SRL	(HL)	CB3E	203,62
SRL	(IX+d)	DDCBdd3E	221,203,XX,62
SRL	(IY+d)	FDCBdd3E	253,203,XX,62
SRL	A	CB3F	203,63
SRL	B	CB38	203,56
SRL	C	CB39	203,57
SRL	D	CB3A	203,58
SRL	E	CB3B	203,59
SRL	H	CB3C	203,60
SRL	L	CB3D	203,61
SUB	(HL)	96	150
SUB	(IX+d)	DD96dd	221,150,XX
SUB	(IY+d)	FD96dd	253,150,XX
SUB	A	97	151
SUB	B	90	144
SUB	C	91	145
SUB	D	92	146
SUB	E	93	147
SUB	H	94	148
SUB	L	95	149
SUB	xx	D6xx	214,XX
XOR	(HL)	AE	174
XOR	(IX+d)	DDAEdd	221,174,XX
XOR	(IY+d)	FDAEdd	253,174,XX
XOR	A	AF	175
XOR	B	A8	168
XOR	C	A9	169
XOR	D	AA	170
XOR	E	AB	171
XOR	H	AC	172
XOR	L	AD	173
XOR	xx	EExx	238,XX

APPENDICE C

RIVENDITORI SPECTRUM

GBC
Viale Matteotti, 66
Cinisello B. (MI)

COMPUTER CORMER
Via Marsala, 6
Sesto San Giovanni (MI)

SAL COM
Via Salaria, 1031
ROMA

CF ELETTRONICA
Corso Vittorio Emanuale, 54
NAPOLI

APPENDICE D

Istruzioni in codice macchina

ISTRUZIONI Codice operativo	FLAG S	 Z	 –	 H	 –	 P	 N	 C
ADC A,r	@	@	–	@	–	@	0	@
ADC HL,s	@	@	–	@	–	@	0	@
ADD A,r	@	@	–	@	–	@	0	@
ADD HL,s	–	–	–	@	–	–	0	@
ADD IX,s	–	–	–	@	–	–	0	@
ADD IY,s	–	–	–	@	–	–	0	@
AND r	@	@	–	1	–	@	0	0
BIT b,r	?	@	–	1	–	@	0	0
CALL pq	–	–	–	–	–	–	–	–
CALL c,pq	–	–	–	–	–	–	–	–
CCF	–	–	–	x	–	–	0	@
(Il flag H assume il valore precedente del flag C)								
CP r	@	@	–	@	–	@	1	@
CPI	@	x	–	@	–	x	1	–
CPD	@	x	–	@	–	x	1	–
CPIR	@	x	–	@	–	x	1	–
CPDR	@	x	–	@	–	x	1	–
(Il flag Z assume il valore 1 se BC assume il valore zero. P/V assume il valore 1 se A =(HL-1))								
CPL	–	–	–	1	–	–	1	–
DAA	@	@	–	@	–	@	–	@
DEC r	@	@	–	@	–	@	1	–
DEC s	–	–	–	–	–	–	–	–
DI	–	–	–	–	–	–	–	–
DJNZ e	–	–	–	–	–	–	–	–
EI	–	–	–	–	–	–	–	–
EX AF, AF'	–	–	–	–	–	–	–	–
EX DE,HL	–	–	–	–	–	–	–	–
EX (SP),HL	–	–	–	–	–	–	–	–
EX (SP),IX	–	–	–	–	–	–	–	–
EX (SP),IY	–	–	–	–	–	–	–	–
EXX	–	–	–	–	–	–	–	–
HALT	–	–	–	–	–	–	–	–
IM 0	–	–	–	–	–	–	–	–
IM 1	–	–	–	–	–	–	–	–
IM 2	–	–	–	–	–	–	–	–
INC r	@	@	–	@	–	@	0	–
INC s	–	–	–	–	–	–	–	–
IN A,(n)	–	–	–	–	–	–	–	–

ISTRUZIONI Codice operativo	FLAG S	Z	-	H	-	P	N	C
IN r,(C)	@	@	-	@	-	@	0	-
INI	?	x	-	?	-	?	1	-
IND	?	x	-	?	-	?	1	-

(Il flag Z assume il valore 1 se B assume il valore zero)

ISTRUZIONI	S	Z	-	H	-	P	N	C
INIR	?	1	-	?	-	?	1	-
INDR	?	1	-	?	-	?	1	-
JP pq	-	-	-	-	-	-	-	-
JP c,pq	-	-	-	-	-	-	-	-
JP (HL)	-	-	-	-	-	-	-	-
JP (IX)	-	-	-	-	-	-	-	-
JP (IY)	-	-	-	-	-	-	-	-
JR e	-	-	-	-	-	-	-	-
JR c,e	-	-	-	-	-	-	-	-
LD(BC),A	-	-	-	-	-	-	-	-
LD A,(BC)	-	-	-	-	-	-	-	-
LD (DE),A	-	-	-	-	-	-	-	-
LD A,(DE)	-	-	-	-	-	-	-	-
LD I,A	-	-	-	-	-	-	-	-
LD R,A	-	-	-	-	-	-	-	-
LD A,I	@	@	-	0	-	x	0	-
LD A,R	@	@	-	0	x	0	-	

(P/V è settato per interrompere il flag di memoria)

ISTRUZIONI	S	Z	-	H	-	P	N	C
LD SP,HL	-	-	-	-	-	-	-	-
LD SP,IX	-	-	-	-	-	-	-	-
LD SP,IY	-	-	-	-	-	-	-	-
LD r,r	-	-	-	-	-	-	-	-
LD s mn	-	-	-	-	-	-	-	-
LD A,(pq)	-	-	-	-	-	-	-	-
LD s,(pq)	-	-	-	-	-	-	-	-
LD (pq),A	-	-	-	-	-	-	-	-
LD (pq),s	-	-	-	-	-	-	-	-
LDI	-	-	-	0	-	x	0	-
LDD	-	-	-	0	-	x	0	-

(P/V assume il valore zero se BC assume il valore zero)

ISTRUZIONI	S	Z	-	H	-	P	N	C
LDIR	-	-	-	0	-	0	0	-
LDDR	-	-	-	0	-	0	0	-
NEG	@	@	-	@	-	@	1	@
NOP	-	-	-	-	-	-	-	-
OR r	@	@	-	0	-	@	0	0
OUT (n),A	-	-	-	-	-	-	-	-
OUT (C),r	-	-	-	-	-	-	-	-
OUTI	?	x	-	?	-	?	1	-
OUTD	?	x	-	?	-	?	1	-

(Z assume il valore 1 se B assume il valore zero)

ISTRUZIONI	S	Z	-	H	-	P	N	C
OTIR	?	1	-	?	-	?	1	-
OTDR	?	1	-	?	-	?	1	-
POP AF	x	x	x	x	x	x	x	x

(I valori dei flag sono determinati dal contenuto del byte al top dello stack)

ISTRUZIONI	S	Z	-	H	-	P	N	C
POP s	-	-	-	-	-	-	-	-
PUSH AF	-	-	-	-	-	-	-	-
PUSH s	-	-	-	-	-	-	-	-
RES b,r	-	-	-	-	-	-	-	-
RET	-	-	-	-	-	-	-	-
RET c	-	-	-	-	-	-	-	-
RETN	-	-	-	-	-	-	-	-
RETI	-	-	-	-	-	-	-	-
RLA	-	-	-	0	-	-	0	@

ISTRUZIONI Codice operativo	FLAG S	Z	-	H	-	P	N	C
RL r	@	@	-	0	-	@	0	@
RLCA	-	-	-	-	-	-	-	-
RES b,r	-	-	-	-	-	-	-	-
RET	-	-	-	-	-	-	-	-
RET c	-	-	-	-	-	-	-	-
RETN	-	-	-	-	-	-	-	-
RETI	-	-	-	-	-	-	-	-
RLCA	-	-	-	0	-	-	0	@
RRCA	-	-	-	0	-	-	0	@
RLA	-	-	-	0	-	-	0	@
RRA	-	-	-	0	-	-	0	@
RLC r	@	@	-	0	-	@	0	@
RRC r	@	@	-	0	-	@	0	@
RL r	@	@	-	0	-	@	0	@
RR r	@	@	-	0	-	@	0	@
RRD	@	@	-	0	-	@	0	-
RLD	@	@	-	0	-	@	0	-
RST 00	-	-	-	-	-	-	-	-
RST 08	-	-	-	-	-	-	-	-
RST 10	-	-	-	-	-	-	-	-
RST 18	-	-	-	-	-	-	-	-
RST 20	-	-	-	-	-	-	-	-
RST 28	-	-	-	-	-	-	-	-
RST 30	-	-	-	-	-	-	-	-
RST 38	-	-	-	-	-	-	-	-
SBC A,r	@	@	-	@	-	@	1	@
SBC HL,s	@	@	-	@	-	@	1	@
SCF	-	-	-	0	-	-	0	1
SET b,r	-	-	-	-	-	-	-	-
SLA r	@	@	-	0	-	@	0	@
SRA r	@	@	-	0	-	@	0	@
SLR r	@	@	-	0	-	@	0	@
SUB r	@	@	-	@	-	@	1	@
XOR r	@	@	-	0	-	@	0	0

Per chiunque consideri il linguaggio macchina un argomento irraggiungibile ed esclusivo patrimonio di pochi tecnici specializzati, questo libro potrebbe costituire la più piacevole delle smentite.
Attraverso un linguaggio estremamente chiaro e partendo dal presupposto che il lettore sia totalmente inesperto dell'argomento, il testo introduce ai segreti del linguaggio macchina dello Spectrum, un argomento in realtà assai lineare e alla portata di tutti, oltre che estremamente utile.

L. 16.500

COD 411D ISBN 88-7056-209-3

GRUPPO
EDITORIALE
JACKSON
James Walsh
IL LINGUAGGIO MACCHINA DELLO SPECTRUM VOL.1
174